30707

National Library of Canada | Bibliothèque nationale du Canada

CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

NAME OF AUTHOR/*NOM DE L'AUTEUR* James Heifetz

TITLE OF THESIS/*TITRE DE LA THÈSE* PL370: A Machine-Oriented Language for the IBM 370's

UNIVERSITY/*UNIVERSITÉ* University of Alberta

DEGREE FOR WHICH THESIS WAS PRESENTED/ *GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE* M.Sc.

YEAR THIS DEGREE CONFERRED/*ANNÉE D'OBTENTION DE CE GRADE* 1976

NAME OF SUPERVISOR/*NOM DU DIRECTEUR DE THÈSE* B[redacted] J. Mailloux

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

DATED/*DATÉ* 1976 October 7 SIGNED/*SIGNÉ* James Heifetz

PERMANENT ADDRESS/*RÉSIDENCE FIXE*
c/o Mr. M. Heifetz,
49 Ternhill Crescent,
Don Mills, Ontario
M3C 2E4

NL-91 (3-74)

INFORMATION TO USERS

THIS DISSERTATION HAS BEEN
MICROFILMED EXACTLY AS RECEIVED

This copy was produced from a microfiche copy of the original document. The quality of the copy is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Canadian Theses Division
Cataloguing Branch
National Library of Canada
Ottawa, Canada K1A 0N4

AVIS AUX USAGERS

LA THESE A ETE MICROFILMEE
TELLE QUE NOUS L'AVONS RECUE

Cette copie a été faite à partir d'une microfiche du document original. La qualité de la copie dépend grandement de la qualité de la thèse soumise pour le microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

NOTA BENE: La qualité d'impression de certaines pages peut laisser à désirer. Microfilmee telle que nous l'avons reçue.

Division des thèses canadiennes
Direction du catalogage
Bibliothèque nationale du Canada
Ottawa, Canada K1A 0N4

The University of Alberta

PL370: A MACHINE-ORIENTED LANGUAGE FOR THE IBM 370'S

by

© James M. Heifetz

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science
Edmonton, Alberta
Fall, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "PL370: A Machine-Oriented Language for the IBM 370's" submitted by James M. Heifetz in partial fulfilment of the requirements for the degree of Master of Science.

Barry J. Mailloux
(Supervisor)

....

Dale H Bent

Date 1976 September 14

Abstract

PL370 is a programming language derived from the general-purpose language ALGOL 68. It is intended mainly for systems programming. It is more difficult to use than ALGOL 68, because one must specify his program more completely, but allows the programmer to avoid expensive generalized features. The design goals are given and the need for yet another language explained. Other languages that have been used for systems programming are surveyed for those features that have been successful and those that have been failures in the past. The various components of the language are considered one by one and the resulting PL370 language is described. The functional requirements of a compiler are given, because a good compiler is essential to a successful language.

Acknowledgements

I hereby acknowledge the important help I have received from others. My wife, Elinor, has been a constant source of inspiration. Several fellow students, including Chris Gray and Chris Thomson, have been of great help in listening to ideas, and in contributing their own. I would also like to thank my supervisor, Barry Mailloux, for his help throughout the period of my research and especially, for his help in the final stages.

Table of Contents

Library Release Form i

Title ii

Approval Page iii

Abstract iv

Acknowledgements v

Table of Contents vi

Table of Figures viii

CHAPTER

1 Introduction 1
- 1.1 Motivation 1
- 1.2 An Overview of the Thesis 5
- 1.3 A Short Glossary of ALGOL 68 Terms 7
- 1.4 Language Design Goals 8

2 Review of Alternative Languages 12
- 2.1 IBM Assembler Language 13
- 2.2 PL360 16
- 2.3 FORTRAN 17
- 2.4 Bliss 19
- 2.5 BCPL 24
- 2.6 C 26
- 2.7 Pascal 28
- 2.8 ALGOL 68 33
- 2.9 PL/S 36
- 2.10 SUE 38
- 2.11 MARY 40
- 2.12 Summary 44

3 Consideration of Various Language Constructs 47
- 3.1 Conditional Clauses 47
- 3.2 Condition Code Handling 53
- 3.3 Loops 56
- 3.4 GO TO statements ....[illegible]
- 3.5 The Meaning of REF 66
- 3.6 Packing of Data 71
- 3.7 Storage Allocation 73

3.8 Register Management 76
3.9 Scope 82
3.10 Addressability 86
3.11 Multiple Values 88
3.12 Linkage Conventions 91
3.13 Operators 95
3.14 Exception Handling 95
3.15 Basic Actions 97
3.16 Declarations 119
3.17 Expressions 124
3.18 Macros and Compile-Time Actions 126
3.19 Separate Compilation Mechanisms 132
3.20 Restrictions on Capabilities of the Language .134
3.21 Support for Languages other than English 135
3.22 Use of Extra Graphic Symbols 136

4 Final Design of the Language 139
4.1 Data 139
4.2 Declarations of Declarers 145
4.3 Declarations of Objects 154
4.4 Scope Rules 158
4.5 Actions 161
4.6 Compilation Structures 169
4.7 Extensibility Features 171
4.8 Run-time Organization 174

5 Implementation 176
5.1 Desired Environment for Program Design 176
5.1.1 Compiler Services 176
5.1.2 Register Management 178
5.1.3 Run-Time Support 179
5.2 Outline of a Compiler 179

6 Conclusion 182
6.1 What was Accomplished 182
6.2 Future Work 184

Bibliography 185

APPENDIX

I Moving and Adding in IBMASM 188

II Sample Linkage Declarations 190

Table of Figures

1. The Design Objectives of Bliss 20
2. Language Evaluation Summary 45
3. States of the World under MTS 92

CHAPTER 1

INTRODUCTION

1.1 Motivation

A major problem in large systems is incorrect linkage between constituent routines (sub-programs) of the system. A common error is that a calling routine and a called routine are written using different conceptions of how the call will be made. The term "linkage convention" is used to include all the details of how a call is made - not what routine is called or what information is passed, but how information is passed. Most systems programmers using IBM machines still program in assembly language, even though it contains few features to help them detect errors in their programs. The incident related below provided the motivation for the development of an assembler replacement.

In early 1974, Barry Mailloux received the ALGOL68C compiler [8] and associated software from University of Cambridge in England. I soon took responsibility for maintaining this system at the University of Alberta. The system included a run-time library written in IBM Assembler [3] for use under IBM's OS operating system. The run-time library had to be converted to run under the MTS operating system in order to be used here. Some of the routines, such as the one for formatting of data before output, are operating-system independent. Others, such as the one called "open", are almost completely different in the two versions. The style of programming used by the original authors in Cambridge was very terse and almost without commentary; there was no documentation about the basic structures of the whole system. I was afraid of being unable to make the necessary new run-time routines because I did not completely know the linkage conventions that they used. This fear was justified several times. For example, about one year after the conversion was supposedly complete, I discovered that the trigonometric functions did not work because they were supposed to return the result in a different register than the one I had used.

The linkage problem is made worse in assembly language because even when you know what the linkage conventions are, you must be very diligent to abide by them. If by mistake, you break a convention, it is extremely unlikely that an assembler can catch this error.

Since I was unsure of the conventions, I wanted to be able to write them down somewhere once and for all and let a compiler enforce the conventions. Therefore, I decided that I wanted to rewrite the run-time library in a better language - one which had a compiler that would catch many more logic errors than an assembler. I then looked around for a language to use.

There is one absolute requirement for any language used to write this run-time library. One must be able to write procedures using the same linkage convention as every other ALGOL68C routine. This immediately eliminates all possible replacements other than PL360[23]. But neither PL360 nor Assembler meets the goal of being able to enforce linkage conventions. In fact, neither the basic assembler language (Assember minus conditional assembly) nor PL360 give the programmer any help at all in doing this. The Assembler also has a macro-processor, and most conventions can be embedded in macro definitions. Although this does not enforce conventions, it greatly reduces the workload on the programmer, because all he need do is remember never to circumvent the few essential macros. PL360 has no macros[1], which makes procedure calling hard-to-write, error-prone, and hard-to-read. Therefore PL360 was not an acceptable replacement language.

The acknowledgement that there was no acceptable replacement for Assembler, which is generally considered a poor language, was not made happily. What was needed was a language that managed the registers for the programmer, especially in the sensitive area of linkage, but could accept some instructions from the programmer on how to do it. There are currently mo such languages.

---

[1] Some versions of PL360 with macros are appearing now, but there does not seem to be any "official" or generally-available version with macros.

Now the idea of updating PL360, to reflect the advances over ALGOL 60 made by ALGOL 68[34], had been around for some time. I decided to design a new, updated version of PL360. Initially, it was called "PL368" - the "60" in "PL360" was replaced by "68" to indicate that the guiding language was ALGOL 68 rather than ALGOL 60. However the "60" in "PL360" is also part of the "360", meaning that PL360 is oriented towards the IBM 360. PL360 does not use any of the new instructions introduced on the IBM 370 series. Also PL360 does not support many of the instructions available on the IBM 360 series, for example decimal arithmetic. I soon decided that the updating of PL360 should also include the capability of using all the instructions on the IBM 360. Later I decided to include all the instructions on the IBM 370[6] as well. At this point, I decided that "PL368" was too unwieldy a name and renamed the language "PL370".

Originally, I intended to make PL370 a true subset of ALGOL 68 by:

30 identifying and eliminating all the very expensive features,

30 cutting out most of the standard prelude,

30 providing a system prelude in which the registers and all hardware operations were made known, and

30 providing a system prelude for each operating system,

in which all its facilities were made available. The task would then be to write a compiler for the subset chosen which could produce code as good as assembly code.

1.2 Overview Of The Thesis

In the latter part of this chapter, the design goals of PL370 are given. Although one goal is to replace assembly language, there is a more ambitious goal behind the stated goals - that there be an area of application , however small, for which PL370 is clearly the best language available. The area chosen was the writing of interfaces. This includes any program that uses more than one convention for the linking of procedures. It also includes run-time systems, which interface a language to an operating system.

Chapter 2 is a study of many of the languages which have been used for systems programming. Although very few of them were designed for this purpose they have been so used and are thus considered. It is important to realize that criticisms in this chapter are only intended to be applicable when the language is used for systems programming. Heavy criticism does not mean that a language is bad, but only that I consider it a poor choice for systems programming. The point of this chapter is to examine the various constructs used in these languages, in order to borrow those features which are good, and avoid those which are bad.

Chapter 3 discusses language constructs again, but by topic rather than by language. Some parts of PL370 have been taken directly from ALGOL 68 without any consideration of alternatives because the ALGOL 68 form seemed clearly superior. These parts are not discussed in Chapter 3. Those topics discussed include both machine-independent general improvements to ALGOL 68 and machine-oriented additions. The topics fall into the general categories of control structures, data structures, actions, and extensibility.

Chapter 4 gathers together all the constructs outlined in Chapter 3 to give a description of the language. This chapter can be read by itself for those interested only in a description of PL370.

Chapter 5 discusses the feasibility of an implementation and what kind of things I would want a compiler to do. The specifications of what a compiler must do are considered an essential part of the language description by some people, and irrelevant by others, who do not wish to restrict the implementor. The discussion of an implementation is necessarily limited because this is a complete topic in itself.

In the rest of this thesis, the reader is assumed to have at least a reading knowledge of ALGOL 68 (or at least its terminology), to know one ALGOL language, FORTRAN, IBM Assembler, and the IBM 370 machine instructions. Also, when discussing any other language, terms generally used in the description of that language may be used without further comment.

1.3 A Short Glossary of ALGOL 68 Terms

mode	- type
proc	- procedure
tag	- identifier (but not of a mode)
unit	- a statement or expression
clause	- a block (possibly valued)
stowed object	- A structure or a row
row	- an array (any dimension)

1.4 Language Design Goals

(1) Procedures

The language must include a complete procedure facility, including parameter passing (unlike PL360 and assembly languages). The programmer must be able to define procedures using any desired linkage convention. A major use for low-level languages is to rewrite programs that were written in a high-level language but have not been compiled as efficiently as required. For any unit of compilation in any language, it must be possible to replace that unit by a PL370 source module. This will be done by writing the unit as a procedure.

(2) Readability

PL370 programs must be fairly easy to read. In the case of a program in one language being re-written in PL370, as decribed above, one should be able to define the desired environment sufficiently well so that the PL370 replacement reads similarly to the original, or at least in a natural manner to a reader of the original language.

(3) Register Management

The programmer must be able to manage those registers he wants to by himself, without being required to concern himself with the others. The set of registers being managed by the compiler must be changeable within a compilation.

(4) Completeness

The language must be able to express any operations which the machine can perform. There must be no cases for which one must give up and go back to the Assembler. This language must be able to replace the Assembler completely.

(5) Safety

The language must be one in which many errors are recognized as such by the language and the compiler. As many types of errors as possible that must be considered logic errors when using an assembler should be moved to the class of syntax errors.

(6) Powerful Data Structures

The programmer must be able to create abstract data structures and operations upon them, as in ALGOL 68, in order to accurately reflect his algorithm. The programmer must be able to use and manipulate any data created by other programs.

(7) Powerful Control Structures

The programmer needs powerful control structures to improve the compactness and correctness of his programs. Good control structures make programs easier to read and to write. They also allow the compiler to make a better translation.

(8) Convenient Representation

The character set needed to represent a program must be simple, but flexible. The most common character set available (those characters available both on the IBM 029 keypunch and on the PN print train) must suffice for any program, but the programmer must be able to take advantage of any additional characters (symbols) available, such as those on the TN print train.

(9) Macros

There must be a macro-processor. Lack of macros is the major reason why PL360 is worse than Assembler. One should be able to use the macro-processor by itself to generate arbitrary text rather than a PL370 program (e.g. as, OS SYSGEN uses the macro-processor in the Assembler to produce JCL programs).

(10) Elegance

The language should show no weaknesses when compared to other systems-programming languages, especially any available on the IBM 360 or 370. When there is a choice of otherwise equal forms, the one that best resembles ALGOL 68 is preferable.

(11) Simplicity

The language should be easy to learn. Knowing ALGOL 68 already should be an asset.

CHAPTER 2

REVIEW OF EXISTING LANGUAGES

In this chapter, several languages which have been used for systems implementation will be reviewed. The languages chosen include some of the best available, or possibly available for the IBM 370. None is very satisfactory; that is why PL370 was designed. Although each language has been approached on its own terms, there are some basic criteria for judging a language. These are presented below, not in any special order.

(1) Can all the appropriate features of the machine be used? Can the language replace the assembler?

(2) Is it easy to learn?

(3) Does it have powerful data-description facilities? (This feature is very important in large systems.)

(4) Does it have good control structures?

(5) Can one have operators for all his data types?

(6) Can one approach the language from different levels?

(7) Can the language communicate with other languages?

(8) Does it have a good macro facility?

(9) Are most input devices (e.g. a keypunch) sufficient for program input?

(10) Are most output devices (e.g. a printer with a PN print train) sufficient for program listing?

(11) Can one use any extra characters available on a printer?

In what follows, any references to "ALGOL" mean ALGOL W or ALGOL 68.

definition: "IBMASM" - the language accepted by the processor IBM calls Assembler F [3].

definition: a "complete" language is one in which any program that can be written in IBMASM can also be written in that language without resorting to the use of machine language in the data.

definition: A "structure" is a collection of data in one object. This term is used in the same way as in COBOL, PL/I, and ALGOL 68.

At the end of each section is a list of the unacceptable features of the language from the point of view of a systems programmer. In fact, some of these features are good features from the point of view of an applications programmer. All examples in this chapter are written in the language being discussed in the section in which the example appears.

2.1 IBMASM

The basic assembly language allows full control over the machine, but is tedious to use because every single machine operation must be specified. The macro language allows one to extend the language and build "higher-level" features, so that the source program becomes more and more legible. Although the macro definitions are often very ugly, they may be used to build almost anything that could be wanted; the restrictions are in the number of continuation cards and parameter length on the macro call. The big drawback to using the macro processor is that to build nice features, a programmer ends up writing more and more of a parser and code generator as part of his macro definitions.

Tags (identifiers, labels, and operators) are restricted to at most 8 characters.

A 'program' is broken into assemblies. There is no inherent semantic meaning to an assembly; it is simply a collection of possibly related statements being assembled at the same time. There is little language-imposed control on references within an assembly: a tag declared anywhere may be referenced from any other statement (after an appropriate USING statement).

{ The idea of the independent subroutine as a compilation unit is completely missing in IBMASM and also in COBOL. Whereas the latter mildly encourages a "run-unit" to be broken into several "programs", assembly programmers seem to be reluctant to break up their assemblies. FORTRAN has better facilities for this than either language. }

There is no check against the misuse of variables in storage. E.g., one can assign a packed-decimal integer value to a binary-integer variable and the move is made without conversion or complaint.

There are too many operators to remember easily. For example, there are 15 different add instructions and 48 different instructions to move data from one part of the machine to another without conversion (see Appendix I).

Procedures with parameters are poorly supported. With the IBM linkage conventions, the actual parameters are possibly scattered in memory. To reference each parameter (without moving it into a more convenient location) by a tag takes much work on the part of the programmer. It also requires at least one register for each parameter. For this reason, high-level languages move parameters to a more convenient location, but the assembler gives the programmer no help in doing this.

An expression must be split into several lines (at least one for each operator in the expression).

The programmer must keep track of which registers are in use; he can accidentally overwrite any value (although an assembler could give warnings).

Worst Features of IBMASM

- no PROCEDUREs
- no IF-THEN

- no IF-THEN-ELSE
- no CASE
- no type checking
- no expressions
- cannot ignore registers
- no initialization in declarations of reentrant routines
- range of a label too large

2.2 PL360

PL360 [23] was designed by Wirth to be "more civilized" than IBMASM. It can be used almost everywhere that IBMASM can. However, because it has no macro facility, it is not always an improvement.

PL360 is not "complete" because it always requires the use of a base register, even when one is not needed by the program.

There is no support for decimal numbers and poor support for strings.

Dynamically acquired storage could be better supported; one must make the identifiers of the data section available for misuse before the storage is actually acquired if he wants the compiler to compute the length of the section, which is the usual case.

Procedures with parameters are even more poorly supported than in IBMASM. The parameter list must be built explicitly by the caller. PL360 also has all the problems of parameter passing mentioned in the section on IBMASM.

Boolean expressions are very restricted; one cannot have both AND and OR in the same expression.

Worst Features of PL360

- no macros
- passing parameters takes too much work
- very rudimentary type checking
- cannot ignore registers
- (practically) no support for decimal numbers

2.3 FORTRAN

FORTRAN [1] was designed to ease programming of arithmetic formulae. Although it was not intended to be used for systems programming, it has been used for that purpose by many people, so it will be considered here. FORTRAN has several features attractive to systems implementors: it is a readily available tool on most computers, often the only alternative to an assembler; arithmetic is very easy to do; it includes several important basic data types (LOGICAL,INTEGER,REAL), arrays of several dimensions, and independent subroutines; and it is usually well-supported in that most subroutines provided by an installation are callable from FORTRAN programs.

On the other hand the language has many defects, which have been documented at length in the literature, so only a brief list will be given here.

There is a low limit (6) on the number of characters in an identifier.

Identifiers need not be declared; thus misspellings are seldom caught by a compiler, and only with difficulty by a programmer.

There are unnecessary restrictions throughout the language: e.g.

- no expressions in the DO loop parameters
- only simple expressions for subscripts
- a restricted number of dimensions in arrays
- the DO loop increment must be positive
- the DO loop control values must be simple identifiers or constants
- the "then-part" of an IF statement must be from a restricted set of statements.

The control structures provided are minimal. There are only CALL, DO loops (but see above), IF (with only one statement in the then-part and no else-part at all), and various kinds of GO TOs. Only numeric labels are provided. The IF statement is so weak that the then-part is almost always a GO TO statement. Because of the DO loop restrictions, otherwise unnecessary variables clutter the program. As a result of all this, FORTRAN programs are very often badly organized and hard to read.

A FORTRAN program cannot communicate with a program written in another language, unless both implementations happen to use compatible linkage conventions.

A FORTRAN program is unable to use any storage other than that acquired at load time or at subroutine entry; i.e., one cannot build linked-lists with pointers at run-time as in ALGOL, IBMASM, and PL360.

There are no pointer variables or structures. Also there are no named constants.

Worst Features of FORTRAN

- no IF-THEN-ELSE
- no IF-THEN
- no CASE
- no structures
- cannot use registers
- cannot communicate well
- weak DO loop
- sub-programs cannot be re-entrant

2.4 Bliss

The authors of Bliss [39] handle very well the problem of designing a product that looks like a high-level language, yet is efficient. They do not really consider communication with programs written in other languages at all, whereas communication is a major objective of PL370. The design objectives for Bliss (see Figure 1 below) are good, but IBMASM meets six of the first seven very well, failing the fifth one. This would make one wonder about the desirability of Bliss, yet there is no doubt that Bliss is actually superior to any assembly language. It seems that

The Design Objectives of Bliss
as ranked by the authors of [39]

(1) space/time economy
(2) access to all relevant hardware features
(3) object code must not depend on elaborate run-time support
(4) control over the representation of data structures
(5) flexible range of control structures (including recursion, coroutines, asynchronous processes)
(6) modularization of a system into separately compilable sub-modules
(7) parameterization, especially conditional compilation
(8) encourages program structuring for understandability
(9) encourages program structuring for debugging and measurement
(10) economy of concepts (involution), generality, flexibility, etc.
(11) utility as a design tool
(12) machine independence

Figure 1. The design objectives of Bliss

more of the design objectives are considered absolute by the authors than they imply. Because PL370 is concerned with inter-language communication, some of the characteristics of Bliss are unacceptable:

a) There is no requirement that the objects supplied for external declarations actually match the declarations.

b) The data are always stored on a stack, which requires some run-time support on an IBM 370 and many other machines.

c) Control over registers is said to be available, but not described in the documentation [39,40], so the control must be assumed to be not very good.

The design of Bliss is very much dependent on the PDP-10 architecture. The IBM 370 is a very different machine from the PDP-10. The PDP-10 is a machine with a single address space of untyped words. Some of the words are in registers and some are in memory. The IBM 370, on the other hand, has four separate address spaces: 3 sets of registers (general-purpose, floating-point, and control), each register being a typed word, and a main memory composed of untyped bytes. Because of the varied data types and alignment restrictions on the IBM 370, it is possible to look at a memory dump and make some intelligent guesses as to where the data are and what the type of each datum is. This has two consequences. First, the typelessness of Bliss, which matches that of the PDP-10 exactly, is inappropriate for the IBM 370. One must have at least four types: one for each address space. Secondly, the interpretation that the use of an identifier, such as 'x', means the address of the object and that '.x' is necessary to get its contents, does not work completely on that machine. If we have ' register x, local y ', then, on an IBM 370, it may happen that ' x = y '; this is not possible on a PDP-10. The expression '.2' could have up to five different interpretations on the IBM 370, depending on context: the contents of memory byte 2, general register 2, control register 2, floating-point register 2(short), or floating-point register 2(long). The usual rule in a language used on the IBM 370 is that one may not ask for the address of a register; this again requires at least a minimal typing system.

Bliss always requires the use of an operator to retrieve the contents of a variable. This leads to many (possibly ugly) uses of the dot operator. A reason for this requirement is the inability to distinguish between addresses and other values in a typeless language. It can be confusing to try to decide whether '.x' is itself an address or another value. With a strong typing system, taking the contents of a variable automatically and only when correct is both easy and desirable.

Bliss has no go to statement. In fact, it has no statements, for it is an expression language. The wealth of control structures is very good, except for the eight different kinds of exit control. It is preferable to require only one exit keyword and to allow labels on groups. For a further discussion on go tos, see Chapter 3.

The co-routine mechanism appears clumsy and hard to understand. Its chief asset is in forcing the programmer to prepare for a co-routine to RETURN. However, one cannot transfer from one co-routine to another by mentioning its name; this unusual mechanism looks error-prone.

Bliss structures appear to be "intelligent arrays". One cannot access a field of a structure by identifier as in ALGOL, COBOL, PL/I, or Pascal. Instead, the programmer "trains" subscripts to behave differently according to which structure is the current map for the object. Note that mapped objects are, in fact, typed objects, even though the particular type of an object may be temporary. A structure may also be thought of as a function, invoked implicitly, which returns a value typed as an address. The programmer has his choice of how each structure is to be allocated. This is flexible, but it encourages the use of several different allocation schemes. Furthermore, this mechanism is fragmentary - each structure must be allocated separately; there are times when one would like to allocate all the local objects at once. In general, Bliss structures seem far inferior to the usual kind, i.e., those in ALGOL, COBOL, or PL/1.

Worst Features of Bliss

- only 1 type, of fixed size
- cannot communicate well
- no labels
- error-prone co-routines
- no structures

2.5 BCPL

BCPL [29] is a general-purpose language. It is primitive in that it has only one type (bits), but a fairly rich control structure gives it the look of a higher-level language. It uses the concept of "mode of evaluation": an expression evaluated in "Lmode" produces the name of a cell; one in "Rmode" produces a value. Normally, expressions on the left-hand side of an assignment are evaluated in Lmode, and those on the right-hand side in Rmode, but the programmer can override this.

BCPL, like Bliss, has a very rigid concept of run-time organization. It cannot completely replace an assembler, and indeed for some applications may be less secure. Since all variables are of type bits, they are all the same size. There is always a stack for local variables and temporaries, and there is no access to registers.

Structures must be simulated using vectors and naming the displacements of each field (like Bliss and FORTRAN); this is an inferior method.

The syntax is different from that of most other languages in many ways, some of them interesting. Some examples are:

a) a multiple assignment statement is provided; e.g.

```
t1, t2, t3 := s1, s2, s3;
```

is equivalent to

```
(t1 := s1, t2 := s2, t3 := s3);
```

b) vectors are created by "vec <expression>", which allocates a vector of the desired length and returns a pointer to it.

c) IF statements have been split into two types, both ugly. CASE statements are called switchon.

d) Declarations of global variables specify their displacements within the common area, not their initial values. For example, " global x = 14 " means that x is the 14th word in the global area, not that x has a value 14. This is confusing because no other language (including assembly languages) has the programmer specify addresses of variables.

Worst Features of BCPL

- only 1 type, of fixed size
- cannot use registers
- no structures
- ugly syntax
- cannot communicate

2.6 *C*

C [30] is a high-level language that is usable for systems programming; for example UNIX [19] was written in C. As in the case of FORTRAN, a routine written in C may call or be called by a routine written in another language - as long as both use the same linkage conventions.

C has a weak type system. It provides *char*, *int*, *float*, *double*, *array*s of ..., *function*s *returning* ..., and *pointer to* ... . However, the system allows much abuse.

- *char* is really *short int* with some character features
- character constants are of type *int*
- *float* is lengthened to *double* almost anywhere, even if this is not wanted
- *int*s and *char*s may be used as pointers
- There is no type *bool*, per se. Boolean expressions return *int*s 1 or 0, and accept any positive *int* as TRUE
- *int* variables may be used as label variables!

Pointers are indicated indirectly. " int *g " defines a variable, g, of type *pointer to int*, which may point at *int*s. Similary, *pointer to pointer to char* is indicated by " char **tag ".

C includes both prefix and postfix unary operators. For instance, " *g++ " means "increment g by one, dereference its original value, and return the variable thus pointed at".

The syntax is a little strange for one used to ALGOL.

- The for statement has the form:

  for (<parameters>) <statement> .

The 3 parameters, separated by semi-colons, are an initialization statement, a completion test, and an incrementing statement (which, hopefully, does increment the variable initialized in the first parameter). Braces are used to bracket compound statements.

- "=" is the assignment operator, even in expressions; "==" is the comparison operator.
- Conditional expressions have the form " E1 ? E2 : E3 ".
- Case statements are called "switchon"; each case is prefixed by "case". The default action after each case is to go to the (textually) next case.

<u>break</u> is provided to escape from loops and case statements. <u>continue</u> terminates the current iteration of the innermost loop.

There is a rudimentary macro facility. "#define" may be used to cause subsequent occurrences of an identifier to be replaced by an arbitrary string. This is useful for parameterization of array sizes (which are determined at compile-time) and for making programs more legible or in a format with which the programmer is more comfortable.

Pointer arithmetic is unusual. Consider:

```
int  *p, *q, i, j
```

Then " p + 3 " is the address of the 3rd int after the one p points at.

" p - 3 " is the address of the 3rd int before the one p points at.

" p - q " is the number of ints between p and q. This means that

```
p - (q = p + 4) == 4
```

is true.

Variables may be created automatically upon function entry or may be static. Registers may be used, but only for local storage, i.e. <u>not</u> for communicating between functions.

<u>Worst Features of C</u>

- inconsistent type system
- cannot communicate
- unconstrained pointers

2.7 <u>Pascal</u>

Pascal [37] was designed to be efficient and easy to teach. Some of its good points are given below.

The programmer may define new data types by presenting a set of values. Encoding the values then becomes the compiler's responsibility. The programmer uses values that have meaning, instead of some code, which he might use incorrectly. Because of the strong system of types, there is even more protection than in a language with a weak system of types: one cannot accidentally use a value from the wrong set.

Arrays may be indexed by any finite data type, not just integers.

Named constants are provided. This is good for two reasons. First, the name may indicate the use of the constant, e.g. 'nmb tape drives = 4'. Second, if the 'constant' should ever change, e.g. two more tape drives were purchased, it is easy to change the value.

There are two kinds of boolean-controlled loops: the kind that tests before the body (and is repeated zero or more times), and the kind that tests after the body (and is repeated one or more times). The latter is theoretically unnecessary, but faster when appropriate.

The with statement allows one to factor out all the occurrences of a record pointer from the source program when accessing several fields in one record; this can reduce the source-program size considerably, making the program easier to read.

Sets are available as a data type. One may have sets of any other data type (short of circularity). Very few languages provide sets explicitly (see SETL), although many problems involve sets of objects, and keeping track of them.

Unfortunately, Pascal has many faults.

The Pascal record with variants seems an inferior concept to ALGOL 68's UNIONs. The Pascal record has an explicit tag field which may be set by the programmer. The programmer is not prohibited from changing this tag to change the interpretation of the bits in the variant part of the record; in fact, there is no prohibition against using field selectors which do not correspond to the variant specified by the tag field. This can easily lead to errors.

The declarations seem confusing; it is not immediately obvious which are the type identifiers and which are the new variables. Looking for lists, as in other languages, does not help; a list may be either a type declarer or a list of identifiers.

Dereferencing is done by a postfix operator. This is hard to read because all other monadic operators use the usual prefix style.

Case statements have a problem similar to declarations; it is easy to confuse a label on an individual case, with a value selector for that case. Labels and scalar values look very similar.

Pascal has certain restrictions in common with other high-level languages. A program cannot access registers or other machine data not in main memory (e.g. CPU id). There is one fixed linkage convention chosen by the compiler; no other may be used. There is thus no way for communicating with programs in other languages. The programmer has no explicit control over word sizes, and very little implicit control.

There is no way to use external objects, and no separate compilation mechanism.

Variables may not be initialized in their declaration. At best this can make a program longer than necessary, and at worst, slow it down tremendously. Some ALGOL W programs take almost 40% of their time initializing variables because each array element must be initialized separately [7].

There is no support for variable-length strings. The IBM360 is not very good with them, but better than Pascal.

One cannot define operators for all his new data types. Nor is there any macro facility to allow concise expressions.

An arbitrary type may be used as an index for an array by naming that type. This implies some kind of table mechanism such as SNOBOL has, if that type is not inherently ordered or dense (e.g. a record type). While this is a nice feature for a general-purpose language, it may require extensive run-time support.

The following items are restatements of criticisms made by Habermann [16].

There are no blocks as in ALGOL; BEGIN-END is used only for procedure bodies. This means that variables can not be made local to a sequence of statements without gathering that sequence into a procedure. One loses either protection or efficiency (and possibly locality of reference). Further there is no own storage as in ALGOL 60, or static storage as in IBMASM, FORTRAN (i.e. COMMON), COBOL, and PL/I. This means that if it is required that a procedure remember its current state and resume from there when called the next time, the control variable(s) must be declared in the containing procedure, becoming susceptible to harmful references by outside procedures which should be unable to reference it.

Conditional expressions have not been included.

A FOR loop may be incremented only by ±1. One may have a WHILE or REPEAT loop with a global control variable and an explicit increment statement, but FOR statements are supposed to eliminate these, being much safer (because the programmer cannot forget to increment his control variable). (Some IBM 370 hardware counts only by one (BCT), but arbitrary increments are routinely used with other looping constructs.)

Worst Features of Pascal

- no block structure
- label range is complete containing procedure
- no initialization in declarations
- no separate compilation
- unchecked variants
- cannot use external objects
- FOR loop increment is ±1
- cannot use registers
- cannot communicate

2.8 ALGOL 68

Although the Revised Report [34] has just been published and few implementation projects have been completed, some of the good features of the language appeared nine years ago in 1967 [33]. This is really a long time in computing science. Some of the goals of the authors are applicable to any language design project.

(a) catch as many errors as possible

(b) [if compiling] catch as many errors as possible at compile-time

(c) allow an optimizing compiler to do a good job

(d) build the language out of a few independent pieces which interact well

(e) remove restrictions wherever possible.

Some of the good points of the language are presented below.

Constant data may be declared, to improve readability or save values; a variable may be initialized via an arbitrary expression in its declaration.

New <u>modes</u> may be defined to better convey the conceptual data. The constructors of modes include pointer to another mode, array of another mode, structure with named fields each of some mode, union of several modes, and procedure returning a value of a specified mode possibly with parameters of specified modes. Unions were introduced in ALGOL 68; they specify a set of modes for a datum, allowing more freedom than specifying just one of the modes, but without the unrestricted chaos of no specification. The mode of the current value is determined (and may vary) at run time. Procedure modes are different from many languages in that they completely specify the modes of their parameters. In a procedure declaration, any parameter which is itself a procedure must specify the modes of its own parameters; recursively, if one of these is a procedure, it, too, must specify the modes of its own parameters. E.g.:

```
PROC  c  =  ( INT x,
              PROC ( REAL ,
                     PROC ( INT , INT ) BOOL
                   ) VOID  p
            ) INT :
     ¢     procedure  body    ¢
```

Actually, most ALGOL 68 data structures could be represented fairly similarly in PL/I, but the ALGOL 68 modes and declarations describe and control the data more accurately and more safely.

A procedure may return a value of any mode.

One may define operators for new (or old) modes. This makes the mode definition facility more useful. The combination of a good name for a mode, a good definition of the mode, and appropriate operators for it, can make the use of data almost as easy as in its "natural" form. Included among the built-in operators are the "op-and-becomes" operators (such as "+:=", called plus-and-becomes) which manipulate one variable using a second datum (e.g., "a +:= b" means "a := a+b" except that "a" is evaluated only once, which is useful if the left-hand-side is a subscripted variable or a PROC REF AMODE).

Control statements with end-markers were first used in ALGOL 68, allowing any expression to be a complete block without begin or end, because it was surrounded by other delimiters. Although not everyone agrees on FI as an end-marker for IF, many new languages do use end-markers. The ability to use a block anywhere a constant is allowed does more than improve readability; it often allows a shorter, faster program.

The for loop is a good construction. Being able to leave out irrelevant and defaulted parts is a real convenience to program writers and readers. The ability to have both a to (until) condition and a while condition at the same time is important. While this construct is an order of magnitude superior to the for loop in ALGOL 60, which is far superior to the DO loop of FORTRAN, the PL/I DO loop is more powerful still.

Worst Features of ALGOL 68

- has some very expensive constructs (e.g. FLEX, PAR)
- confusing use of REF
- cannot use external objects
- cannot use registers
- cannot communicate
- no separate compilation

2.9 PL/S

The motivation for PL/S[1] is similar to that for PL370. Some people at IBM wanted a systems implementation language as similar as possible to PL/I. PL/S is used for the writing of new IBM systems software.

The major differences between PL/I and PL/S seem to be:

(a) data attributes

PL/S has integers, pointers, bytes, bits, and structures. There may also be other types. Integers may be

---

[1] PL/S is a secret of IBM's. Details presented here are deduced from listings of VS systems, but no authoritative definition has been made public.

FULLWORD, HALFWORD, or n BYTEs long. Pointers may be FULLWORD, 1 BYTE, or 3 BYTEs long. BYTES and BITS may have any non-negative length specified. Fullword integers and fullword pointers may be located in registers. Objects in memory may be explicitly aligned on byte, word, or doubleword boundaries (plus an offset, if desired).

Many of the PL/I attributes are not included. Probable omissions are PICTURE, COMPLEX.

(b) procedure entry and exit

PL/S may use standard linkage conventions, not those PL/I uses.

Some of the problems with PL/S are:

(1) "=" is used both for assignment and for comparison.

(2) There is no CASE statement (assumed since PL/I lacks one).

(3) Pointers are unconstrained; they may be used as if pointing at any kind of structure.

Worst Features of PL/S

- unavailable
- unconstrained pointers

2.10 SUE

SUE [9] was designed for writing an operating system. It can be regarded as a systems-implementation variant of Pascal.

Perhaps the most unique feature of SUE is the way that systems are built out of programs. Much as in COBOL, each procedure has a DATA block, a CONTEXT block (optional), and a PROGRAM block. The DATA block identifies all procedures conceptually local to the current one, and declares all types and data shared between local procedures and/or the current procedure. The CONTEXT block defines shared constants, types, macros, and machine-dependent information. The PROGRAM block contains the executable statements plus data not shared with any of the local procedures. A system consists of a tree of procedures (as in ALGOL), but each is defined independently; they are hooked up by the mechanism of each procedure declaring the names of its local procedures. In the current implementation, all the parts of a system must first be filed, then the compiler is given a list of names of compilation blocks; e.g.

```
DATA MAIN
DATA SON1
PROGRAM SON1
DATA SON2
DATA GRSON21
PROGRAM GRSON21
PROGRAM SON2
PROGRAM MAIN
```

SUE attempted to unify expressions by allowing a list anywhere a single element was acceptable. Other languages restrict lists to subscript, parameter, and possibly for-loop control positions.

SUE has loops of indefinite repetition. Unlike the usual while loop, SUE's cycle ... end may have any number of termination tests and a test may appear anyhere in the loop that a statement could.

The implementation features of SUE are:

a) The ability to not test for range errors during assignment, selection, and subscripting.

b) The attributes fast, register, aligned, and absolute. fast may make a datum bigger (i.e. at least a halfword) so that it may be accessed faster. register means that a datum is located in a register. aligned specifies the alignment of a datum. absolute specifies the logical location of a datum (i.e. the unrelocatable virtual address).

c) Macros may be used instead of procedures to get open subroutines. .

d) The "inline" built-in function can generate machine instructions in-line.

e) The programmer may access an object of any type as though it had any other type by specifying that other type.

f) There are "group" types. These are like records, but the fields are unaligned, and the group can be treated as a single object (ignoring the field boundaries). These are useful for machine-defined structures (e.g., PSW, CSW, CAW).

Unfortunately, SUE is designed for a one-linkage-convention system. One cannot call other routines unless they use the same linkage conventions as the object programs emitted by the compiler. There is no facility for error recovery, although some could be provided if there were appropriate procedures defined in the global environment.

Worst Features of SUE

- cannot communicate

2.11 MARY

MARY [26] is a machine-oriented higher-level language which claims to be a superset of ALGOL 68. In many ways it is an ideal combination of higher-level features and machine-oriented features.

It has the powerful mode-definition facility of ALGOL 68. By properly describing the data, many errors can be avoided.

The concept of mode is expanded to include the "status" of a datum, where status indicates whether the datum is variable or constant. For non-references, this is straightforward. For references, the status applies to the pointer (can one change it to point at a different datum or not?). The mode also includes the status of the datum being referred to (can it be changed *via this reference*?). Unfortunately, there is a slight ambiguity in usage: a VAL INT is a constant integer, but a REF VAL INT refers to an integer which may be a variable. For example, after

```
INT j := 3;
REF VAL INT  k = j;
  j := 7;
```

k refers to j, whose value is now seven.

The concept of rows is better for systems programming than that of ALGOL 68; the descriptor is separated from the elements. A vector of 'n' contiguous elements, each of mode M, is a datum of mode STRUCT(n)M, which is a structure with 'n' fields of mode M, but is selected from by subscript rather than by field name. This corresponds to a vector in most implementations of FORTRAN, except that the lower bound is usually 0. For rows whose size is determined at run time (as in ALGOL), one needs a descriptor; in MARY this is an object of mode ROW M. This object consists of a reference and a length count for the STRUCT(n)M being referred to. A STRUCT(n)M of the correct size may be generated dynamically at run time. Thus the programmer may use descriptors exactly when he needs them. Note that, since the descriptor and the elements being described are different objects, their statuses may differ.

MARY has added several good looping constructs. These are based on letting the control variable pass through a set of noninteger values: either references or SET values (see below). One may let a reference pass through all the possible references to the elements of a row, rather than having an integer pass through the values 0 to the upper bound of the row; Rain[27] claims that the former way is much more efficient. Since the row being traversed may be a row display, the for-list of ALGOL 60 is available in this format.

MARY has borrowed programmer-defined types, calling them SETs, and powersets thereof, calling them MASKs, from Pascal. These are valuable for readability; they eliminate most nonarithmetic uses of integers and catch many errors in mixing modes that are not detectable when integer codes are used. Support for these modes includes a labelled case statement, which chooses the appropriate value for SETs and all appropriate values for MASKs; and FOR loops in which the control variable assumes either all the possible values of a SET mode or all the values thereof in a given MASK variable.

The mode MASK CHAR is especially useful for lexical analysis. There are pre-defined identifiers (of mode VAL MASK CHAR) CHARSET, ALPHA, and NUM which include all the characters valid in a MARY program, all the letters, and all the digits, respectively.

MARY has also borrowed ranges from Pascal. One may define a FIXED or FLOATING mode which consists of a specified range; objects of this mode may then be allocated in the smallest necessary storage. This method of specifying precision is portable, unlike ALGOL's LONGs and SHORTs. However, there is a problem with mixed-precision arithmetic. FORTRAN and ALGOL disallow it - one operand always is converted (explicitly or implicitly) to the precision of the other. But when there are many modes of similar precision as in Pascal and MARY, there are problems of interpretation [16]. For instance, in

```
MODE A = FIXED(1:9);
MODE B = FIXED(2:10);
A x;
B y;
...
x+y
```

is "x+y" allowed, and, if so, what is the mode of the expression? What about "x+1"?

MARY includes UNIONs as in ALGOL 68. This is not like REDEFINES in COBOL or PL/I, where a cell may contain a value of one mode or another; but rather a pointer which refers to an object of one of several different modes, and a mode indicator. It is required that UNIONs must work in a parallel processing environment, where a task may be interrupted at any time.

MARY requires a run-time system to support the object code. In particular, support for a stack, a heap with a garbage collector, and parallel processing are required in the run-time system. There seems to be no way to define external procedures or have partial compilations of a program. Thus one cannot communicate with programs written in other languages.

Many ugly spots in the language have not really been removed; they have just been hidden in the "library prelude". Consequently this prelude must be a mess which could not be implemented in MARY - another language is still needed.

Worst Features of MARY

- cannot use registers
- cannot communicate well

2.12 Summary

The following is a chart evaluating the languages against the 11 criteria mentioned at the beginning of this chapter. The weights and ratings are strictly my personal opinion. The weights are on a scale from 1 to 10 by 1. Each language rating is on a scale from 0 to 5 by 1/2; "+" is used as an abbreviation for 1/2 in the table. Each total is the inner product of the weight vector and the rating.

Ratings of the languages under the 11 criteria

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	TOTAL (weighted)
weight	10	7	10	9	4	1	10	8	3	3	2	335
BCPL	2	3	1	3	1	2	1	0	5	5	1	126
FORTRAN	2	4	2	2	1	2	2	0	5	5	1	144
C	2	2	4	3	2	2	2	2	1	1	1	155
Pascal	2	4	5	5	1	2	0	0	5	5	1	181
ALGOL 68	2	3	5	5	5	3	0	0	3	3	3	183
Bliss	4	3+	3+	4	2	2	2	0	5	5	1	198
PL360	4+	3+	3	3	2	2	4	0	5	5	1	209
MARY	4	4	5	5	4	3	1	0	5	5	1	224
SUE	3	3	5	5	1	2	3	4	5	5	1	246
IBMASM	5	3	4	1	5	2	5	4	5	5	1	256
PL/S	4	3	5	4	1	2	5	4	5	5	1	267
PL370	5	3+	5	5	3	5	4	5	5	5	5	315

Figure 2. Language Evaluation Summary

The following major deficiencies were found among the languages reviewed:

- no PROCEDUREs
- no IF-THEN
- no IF-THEN-ELSE
- no CASE
- weak loop construct
- ugly syntax
- FOR loop increment only ±1
- no type checking
- only 1 type
- inconsistent type system
- unconstrained pointers
- no expressions
- no macros
- passing parameters takes too much work
- (practically) no support for decimal numbers

- no structures
- cannot use registers
- cannot ignore registers
- cannot communicate (well)
- unreasonable restrictions
- no labels
- error-prone co-routines
- no initialization in declarations
- no separate compilation
- range of a label is too large
- unchecked variants
- cannot use external objects
- has expensive constructs
- confusing use of REF

CHAPTER 3

CONSIDERATION OF LANGUAGE CONSTRUCTS

This chapter presents the constructs of PL370 and explains why these particular constructs were chosen. Some constructs are presented by paradigm and some are presented in an extended BNF. The BNF is extended by the use of:

{ x | ... | y }	means choose <u>one</u> of x or ... or y.
{ x }*	means 0 or more occurences of x.
[x]	means { x | } .

3.1 Conditional Clauses

Since one goal is compatibility with ALGOL 68, three conditional forms are obvious:

(a) The standard ALGOL 68 IF clause:

```
  IF  <boolean-clause>
   THEN  <clause>
 { ELIF  <boolean-clause>  THEN  <clause> }*
 [ ELSE  <clause> ]
  FI
```

(b) ALGOL 68's indexed case clause:

```
 CASE  <integer-clause>
  IN  <unit>    { , <unit> }*
[ OUT  <clause> ]
 ESAC
```

This ignores OUSE, which is rarely used in ALGOL 68 and should never have been invented.

(c) ALGOL 68's conformity case clause for manipulating unions:

```
 CASE  <datum-of-a-united-mode>
  IN
     (<mode-indicant> <new-tag>):  <unit>
   { , ( <mode-indicant> <new-tag> ) :  <unit> }*
[ OUT  <clause> ]
 ESAC
```

There are several other forms or modifications worth considering.

(d) Dijkstra's IF statement with guarded commands [12].

```
IF
[     <boolean-unit> -> <clause>
 { || <boolean-unit> -> <clause> }*
 ]
FI
```

This form is different from most in that it is non-deterministic: any one clause may be selected, provided its guarding booleam-unit is TRUE. Also, Dijkstra does not provide for any ELSE case; it is an error if all the guards are FALSE.

(e) A more likely candidate is a deterministic version of the above. There is no point claiming a construct is non-deterministic unless it is. For those who like Dijkstra's proof methods, they will still apply; that is, a proof which assumes no determinism in the order of evaluation of the guards and proves that a program works is not invalidated by determinism. This form is a variant of the COND function in LISP:

```
CASE
  IF  <boolean-unit>  THEN  <clause>
{ IF  <boolean-unit>  THEN  <clause> }*
[ ELSE  <clause> ]
ESAC
```

An alternative first keyword could be FIRSTCASE (to indicate the determinism), FCASE (shorter), COND (like LISP), or SEARCH. Another form for this clause is:

```
      CASE    [ <M-unit> ]
        IN
          <M-unit> : <N-clause>
        { <M-unit> : <N-clause> }*
      [ OUT  <N-clause> ]
      ESAC
```

In the case where there is an <M-unit> after CASE, this value is compared with the other values; the first matching <M-unit> selects the clause to elaborate. In the case where there is no <M-unit> after CASE, TRUE is assumed, i.e. the existing <M-unit>s are <boolean-unit>s. For those who are unhappy with ELIF, this form is better.

(f) Array style notation.

One could change the syntax of the case clause drastically to make it look like an array reference. I.e., instead of:

```
      CASE  e  IN  c1, ..., cn  OUT o ESAC
```

one could say:

```
      ( c1, ..., cn ) [ e | OUT o ]
```

with exactly the same semantics. This form has the advantage of increasing "uniform reference" (for those who believe that this is an advantage). The disadvantage is that this is so uniform as to confuse even a compiler and probably often confuse the programmer. There is usually a difference in semantics between a CASE clause and a row-display; the former elaborates only the chosen case whereas the latter elaborates every element. A resolution of the problem for this form is to not guarantee elaboration of all the elements of a row display (and their side effects) at any time, and then say that any row or structure from which an element is immediately selected, is effectively a CASE clause. A second reason for mentioning this form is that it naturally gives the idea of allowing an OUT part for any subscript. This allows the programmer to recover gracefully from subscript range errors and, in those cases where not all subscripting is being checked, to neatly indicate the operations to be checked.

(g) In some situations, one wishes to perform some action after the case clause, but only if some unit has been selected, i.e., the OUT clause was not elaborated. The programmer then usually re-evaluates the conditions under which one of the cases will be selected. This expression can be complex; why bother when the work has already been done? An optional clause can be allowed after (or before) the individual cases; it would be executed only if the OUT clause is not.

```
CASE  <case-chooser>
[ BEFORE  <void-clause> ]
   IN <cases>
[ AFTER  <void-clause> ]
[ OUT <clause> ]
ESAC
```

This construct means:

```
Calculate the address of the case to choose.
If it is not the OUT clause,
 THEN
    Execute the BEFORE clause
    Execute the chosen case -
        remember its value.
    Execute the AFTER clause
    Return the remembered value
 ELSE
    Execute the OUT clause
FI
```

(h) Although ALGOL 68 case clauses work with codes from 1, incrementing by 1, many routines return codes from 0, incrementing by 4, to allow use of a jump table (i.e. a case construct). PL370 must be able to use these codes without modification. Forcing the user to write:

```
CASE return code DIV 4 + 1  IN ...
```

just to fit ALGOL 68 semantics is not acceptable. Some alternatives are:

(1) `CASE   rc FROM 0 BY 4   IN ...`

This indicates that cases are to be counted first - 0, second - 4, third - 8, etc., rather than the usual 1, 2, 3, etc. If, at run-time, rc is not a multiple of 4, then the program will branch to some unexpected location.

(2) Include in the language a special mode RC which a

procedure could return and which would be the appropriate subrange of WHOLE. Then the CASE construct would automatically work correctly.

(3) Include a subrange mechanism in the language. Then the user could define RC himself or actually list the possible return codes with the procedure; e.g.:

```
PROC p ( ...  ) INT(0,4,8,12)
/* INT(0,4,8,12)  is  intended  to  mean   that
subrange  of  INT including only the integers 0,
4, 8, and 12 */
```

(i) Sometimes, it happens that the action of one case is a subset of that for another case. As usual, the programmer would like to be able to share the common code. In ALGOL 68, the programmer must repeat the code or gather it into a procedure which two or more cases then call. In PL370, he could repeat the code safely by using a macro. But what about sharing without expensive overhead?

One can use very cheap local procedures wherein the overhead is just BAL - BR. However, they can be nested easily only as far as the programmer prepares for by explicitly using different return registers. Also, if there is any nesting, then the overhead increases significantly.

One could treat each case like an implicit procedure. The case chooser could calculate the address of "ESAC" and each case could branch there (like a COBOL paragraph). Then each case could be invoked by name and called. For example:

```
CASE ...
IN ....,
   ... docase(3)    ¢   performs case 3
```

A very simple but less flexible approach is to have the programmer order his cases so that case j includes case j+1 as its last action. Then case j may include case j+1 very simply, by doing nothing. Instead of going to "ESAC", control just falls through to the next case. For example:

```
CASE i
IN    case 1  FALL
      case 2  ,
      case 3  ,
      case 4  FALL
      case 5  FALL
      case 6
ESAC
```

3.2 Condition Code Handling

On the IBM 370 series, conditional execution is done by setting a special register called the "condition code" in one instruction and testing it in a subsequent instruction. This register is 2 bits long; therefore, it has four different possible values. The setting instruction and the testing instruction need not be contiguous, nor need there be a 1-1 correspondence. In most high-level languages, conditional execution is performed by IF-THEN-ELSE statements or CASE statements. The IF-THEN construction covers most actual uses of the condition code, but the machine hardware is more flexible than that and can be used more efficiently in certain cases.

The first optimization to make is to avoid an explicit code-setting instruction in the cases where a previous instruction has set it as a side effect. In most cases this could be done by the compiler but it would have to be very intelligent. The programmer could specify when to use the existing condition code, as in PL360, but this requires him to know which machine instructions will be generated from each source language costruct, and how they set (or don't set) the condition code. At this time it seems more feasible to train the programmers than the compiler, so I would opt for the second approach. This allows the following shortcut:

```
r1 +:= b; IF r1 > 0 THEN a ...
```

may be replaced by

```
r1 +:= b; IF > THEN a ...
```

A compiler might be able to optimize:

```
IF (r1 +:= b) > 0 THEN ...
```

but probably very little more complex.

An IF statement allows only a 2-way branch. In some cases, a 3-way or 4-way branch could be used to advantage. The FORTRAN arithmetic-IF is an example of the former. These cases are most naturally handled by a case statement. A form such as:

```
CASE  condition code  IN
< : <unit>,
= : <unit>,
> : <unit>,
overflow : <unit>
ESAC
```

could handle these more complex cases. Once condition symbols such as ">" and "<" are introduced, it would be desirable to have a way to define new ones, e.g.:

```
CONDITION   +ve = 2,
            mixed = 1  AFTER  tm;
```

Then condition variables would seem natural, e.g.:

```
CONDITION  cc;
 ...  ;
cc := condition code;
 ...  ;
IF  cc  THEN  ...   FI
¢ i.e.  IF  cc = condition code  THEN  ...
```

We then have a new basic type, *condition*, which can be assigned, compared, and used in if- and case-clauses. Although some basic constructs using this type are moderately expensive, this is because of the machine definition; Assembly programmers needing the construct simply use many instructions.

One also needs *sets* of *conditions* (called masks by IBM). E.g.:

```
MASK   ≥ = 0  OR  2,
       ≤ = 0  OR  1;
```

3.3 Loops

In the course of this research, over a dozen different loop forms have been considered. Why all this activity when ALGOL 68 has only 1 loop form? Firstly, to accomodate the hardware instructions built for looping (which, unfortunately, hardly ever match a loop as general as ALGOL 68's). And secondly, to provide additional safety and efficiency. Although some forms of the ALGOL 68 loop don't make sense (e.g. "FROM 7 DO --- OD", "BY 35 DO --- OD", etc.), it is much easier for the programmer to remember the rule that each part of the loop clause is optional (except the loop itself).

3.3.1 Improving Safety and Convenience (machine-independent)

3.3.1.1 Location of Boolean Test

In Algol, the test for termination is always before the loop body. In Fortran, it is always afterwards. Some other languages test anywhere the programmer requests; PL370 allows this because it has an exit statement.

e.g.

```
WHILE  s1; a > b  DO  s2  OD              is equivalent to:
DO  s1; IF a > b THEN EXIT FI; s2  OD        and to:
DO  s1; EXIT IF a > b; s2  OD
```

The exit statement provides sufficient power for terminating any loop. However, some people may be more used to, or may prefer, the while-clause. One way to make the while-clause more useful is to allow it elsewhere than the top of the loop. One possible place is just before the OD. For example:

```
DO   ....    WHILE  <condition>  OD.
```

One could also allow the WHILE in the middle of the loop. This would require the addition of a new word to terminate the condition or a requirement that it be a closed clause. For example:

```
DO s1 WHILE ( a > b )    s2    OD
```

This is equivalent to the three examples above, and does not require a label, but works only at the same nest level. I consider this form of the while-clause inferior to the exit statement.

3.3.1.2 Choice of Keywords

Allow " UNTIL <condition> " as an alternative to " WHILE ¬<condition> ".

Users of languages other than ALGOL 68 may be upset at the size of the keyword "FROM". The familiar assignment symbol should be available in this place, as it is in most languages.

Another popular keyword, especially with UNTIL, is "REPEAT". This can be considered as a synonym of DO. E.g.:

```
REPEAT  s  UNTIL  b  OD
```

is equivalent to:

```
DO  s  WHILE  ¬b  OD
```

3.3.1.3 Control Variables of Modes Other Than INT

Programmers often write loops controlled by variables of types other than integer. In general, a plain while-loop is more verbose and more error prone than the same loop controlled with a for part. It is, therefore, desirable to extend the for loop to control incrementing and testing for any mode.

For any finite mode, this is easy. In PL370, all finite modes are ordered. So, one could allow:

```
FOR i := <value1>
[ BY <integer-unit> ]
[ TO <value2> ]
DO  ...   OD
```

Here <value1> and <value2> are values of the same mode, and the by- part specifies how many values are to be skipped. For example, " BY 2 " would use every second value.

For an arbitrary mode, the compiler must be told how to update the control variable. This can be done in two ways:

(i) Have the user define "+" for the desired mode and INT. Then the regular form (as in shown above) could be used.

(ii) Have a NEXT clause rather than a BY clause. This would be an expression to be evaluated at the end of the loop and put in the control variable. Thus:

```
FOR i:=1  TO n   NEXT i+j
```

is equivalent to

```
for i:=1  step j  until n
```

in ALGOL 60. This will be more palatable than (i) if the reader does not like "+" being used where it is not even remotely related to addition (although it may be incrementing).

3.3.1.4 Traversing Multiples

A classic horror story concerns the implementation of the array assignment statement in the IBM PL/I-F compiler. Consider the statement " A = 0 ", where A is an array of 1 dimension. This statement is internally expanded to:

```
DO I = LBOUND(A,1)  TO  HBOUND(A,1);
   A(I) = 0;
   END;
```

The catch is that when subscript checking is on, the second line invokes an unnecessary check every time it is executed. PL370 must eliminate as many of these checks as possible at compile-time. Another consideration is that at each pass, the effective address is computed as " @A(0)+I*(size of one element) ", or possibly in a worse way. Why should the loop control variable be the subscript when it is really the address of the element we are interested in? If the control variable is an address, then the expression above can be replaced by a single addition. What is indicated is a loop with semantics similar to:

```
FOR REF INT p:=@a[LWB a]
```

```
      BY (size of one element)
      TO @a[UPB a]
      DO
        /* e.g. */  $p := 0
     OD
```

This could be done by requesting that the control variable possess, one at a time, each of the elements of an array. For example:

```
     FOR i  IN a
     DO  i := 0  OD
```

In the DO part, 'i' plays the role of '$p'; that is, an assignment to 'i' assigns to the currently selected element of 'a', while the true control variable (the address of the element) is unaffected.

This construct can be extended to cover any multiple such as arrays of any number of dimensions and sets of anything. Example:

```
     FOR colour  IN (blue,green,red,yellow)   DO ...
```

Chris Gray [15] has proposed allowing several control variables. E.g.:

```
     FOR i  IN a,    j  IN b
     DO  i := j  OD
```

would mean:

```
     (INT o1=LWB a-1,
          o2=LWB b-1,
          d1 = UPB a - LWB a-1,
          d2 = UPB b - LWB b-1;
      FOR i  TO min(d1,d2)
      DO a[i+o1] := b[i+o2]
      OD)
```

Note that the loop stops as soon as 'a' or 'b' has been exhausted. This is similar to:

```
FOR i,j  WHILE (i IN a)  AND  (j IN b)
```

3.3.2 Improving Efficiency (machine-independent)

3.3.2.1 Specification of UP or DOWN

Given a loop such as:

```
read(i,j,k);
FOR x FROM i BY j TO k DO ...
```

there is no way of foretelling the direction of the loop. Even with values for i, j, and k, it may not be obvious which direction the loop should take. Three possible rules are:

(a) Up if k > i ; Down if k < i ; do once if k = i

(b) Up if j > 0 ; Down if j < 0 ; error if j = 0

(c) Up if j > 0 ; Down if j < 0 ; forever with x=i if j = 0

ALGOL 68 uses the third rule [34 (Section 3.5)]. Those with a finite bank account will probably prefer the second. In any case, it is fairly expensive to determine direction at run-time. Therefore, the programmer should be able to specify at compile-time the direction the loop will take. If he does not, and the sign of the by-part is unknown at compile-time, then the compiler must generate extra code to determine direction. Since the purpose of this construct is code reduction and increased speed, the assertion will not be checked; if the programmer specifies " UP BY b ", and 'b' is negative at loop entry, then the loop will iterate until an overflow occurs.

The first proposal below is machine-independent in the sense that almost every machine has registers, but machine-dependent in that the registers useful in a loop differ from machine to machine. A more portable way is to let the compiler manage the registers not needed for linkage.

3.3.3 Improving Efficiency (machine-dependent)

3.3.3.1 Restricting Loops to Use BXLE or BXH

The form:

```
FOR <general reg>
  FROM <int unit> [ UP | DOWN ]
  BYTO <reg pair> [ := (<int unit>,<int unit>) ]
  WHILE <condition>
DO  <clause>  OD
```

and the form:

```
FOR  <tag>
  <beta> <single register>, <double register>
  FROM ...
```

This gives the compiler some registers to use for the loop. <beta> could be "USING" or "WITH" or ",". The second form could be extended to " ... <beta> REGISTERS ... ", meaning that the compiler should store the control values in registers and pick the registers itself.

3.3.3.2 Restricting Loops to Use BCT

If the for-, from-, and by-parts are omitted, then we have:

```
TO <int unit> [ WHILE <condition> ] DO ...
```

In this case the BCT instruction can be used. Indeed, this is almost what that instruction was designed for. But, if the programmer wants access to the counter in his loop, he would usually say:

```
FOR <tag> TO <int unit>.
```

However, using ALGOL 68 semantics, this means that the counting starts at one and increments. If the programmer needs the counter, doesn't care about the order, but does want the efficient BCT, then he must say:

```
FOR <tag> FROM <int unit> BY -1 TO 1 DO ...
```

This could be sweetened to:

```
TO <int unit> COUNT <reg> DO ...
```

Note. This proposal conflicts with another proposed use of the keyword "COUNT" to support BCT as a primitive (See the section "Basic Actions").

3.4 GO TO Statements

Ever since Dijkstra's famous letter [13] condemning go to, people have been suggesting strong curbs on its use.

The extreme (as shown by Bliss) is to forbid it entirely. Of course, one cannot forbid branching; but one must provide alternate stylized access to this low-level concept. Old examples are if clauses, case clauses, and loops. A newer construct is exit, which is more primitive than the others in that it is just a glorified go to, and identifies not a target location but a whole range to be left. It also explicitly includes all the overhead associated with block and procedure exit. Bliss went so far as to remove labels, but that has been a failure [40].

The loop clause and go to are the only constructs in ALGOL 68 which cannot return a value. In Bliss, loops have been given values. This means that the removal of go to allows every construct to always have a value. This makes Bliss work as an expression language. If it were desirable to have an expression language, then this would be a strong argument in favor of removing go to. However, recent work by Gannon[14] has indicated that statement languages are better because programmers make fewer mistakes. Therefore, PL370 is a statement language and this argument cannot be used.

Although go to may be harmful, its lack may, at times, be more harmful (see [20]), unless there is an adequate replacement. If clauses, case clauses, exits, and loops are almost enough, but not quite. One form that has been discussed is a proposal by Zahn given in [20]. The form is:

```
LOOP UNTIL <event.1> OR ...  OR <event.n>:
   <clause.0>
REPEAT
THEN   <event.1> =>    <clause.1>;
         .
         .
         .
       <event.n> =>    <clause.n>
FI
```

<clause.0> is repeated until one of the <event>s is encountered, then the corresponding <clause> is executed. A non-repeating form could use BEGIN - END instead of LOOP - REPEAT.

Another possibility is simpler and less powerful:

```
    TO FIND <tag>          <clause>
```

The semantics of the prefix are that any assignment to <tag> in the <clause> fulfills its purpose, and so, it is followed by an immediate exit from the <clause>.

Can the go to be eliminated? Although alternative constructs can cover most cases, no one has found a set of control structures that is always guaranteed to replace go to without degrading the program in any way. On this ground, it seems unwise to go to the trouble of removing it. Secondly, it must be remembered that it is easy for programmers to fabricate control structures with macros, and if statement labels are included in the language, a GOTO macro would be easy to write. Furthermore, it would surely be used. A go to is less harmful if the compiler knows about it, and the construct should be included for this reason. In conclusion, go to can be eliminated only if statement labels are eliminated. Since statement labels are not needed by any other construct (the only other thing they can be used for is in treatment of instructions as data by basic actions), go to can be eliminated.

3.5 The Meaning of REF

In standard languages, such as ALGOL 60, FORTRAN, and COBOL, there are no pointers at all. Also, there are no constants, only initialized or uninitialized variables. In ALGOL W, references (pointers) are introduced, but are restricted to point only at records, which are distinct from references, so that the world is partitioned into nonreferences and references to nonreferences (although a nonreference may contain a reference). In PL/I, any variable may be BASED, and thus a pointer may point directly at another pointer. Even in cases where pointers point only at structures, as in ALGOL W, it may appear otherwise to the reader of a program. Consider, for instance, the PL/I variable COUNT which is in full

" P -> S.Q -> T.R -> W.COUNT ".

If Q, R, and COUNT are unique, then the previous phrase may be abbreviated to

" P -> Q -> R -> COUNT ".

In all these languages, the use of a variable requests either the name (address) or value (contents) of that variable. It is determined by context which one is desired. As long as the rules are simple, programmers have little trouble.

Languages such as BCPL, Bliss, and ALGOL 68 are quite different in that the destination (LHS) of an assignment, as well as the source (RHS), may be an arbitrarily complex expression, possibly involving procedure calls. Also, a pointer may point directly at another pointer. The programmer may have trouble deciding what a given phrase means or how to say what he wants. The three languages attempt different solutions. In Bliss, the name is always taken and all dereferencing is explicit; this is probably the simplest and the ugliest solution. In BCPL, there are two "modes of evaluation", Lmode and Rmode. Rules specify which mode is used for a given sub-expression. This usually allows BCPL to take the name when appropriate and take the value when that is appropriate; the programmer may explicitly specify which mode to use when he wishes. In ALGOL 68, the name, if any, is taken, but there is a complicated set of coercion rules which try to minimize the amount of writing the programmer must do. In all three however, the programmer may be confused as to the conceptual modes being handled, because Bliss and BCPL have only one mode, and, in ALGOL 68, declarations sometimes leave out a REF.

ALGOL 68 attempted to solve the name-value problem by considering a name to be a different kind of a value: when talking about ALGOL 68, one does not say "the value of x is 3", but rather, "the value, x, is 3". The mode of a name which refers to an integer is REF INT, different from the mode, INT, of an integer value. However, being "economical" with concepts, it built data security and pointers out of the same mechanism, thus making a useful separation and a bad union. The result is an improvement over all preceeding languages. All the kinds of variables previously available are available as well as some more kinds of variables and some kinds of constants. Suppose 'x' is a datum of mode REF M, where M is any mode. At any time, 'x' may be changed to refer to a different M object; thus 'x' is a "variable". However, one may make a read-only copy of what other languages call "the value of the variable" by saying 'M xx = x;'. This causes 'xx' to be an identical copy of the M object which 'x' referred to at the time of the definition of 'xx'. This is important because, in other languages, people do make assignments to variables which were supposed to be 'constant', causing real errors.

MARY attempts to better ALGOL 68 by separating data protection from indirection. To the mode is added the "status", which conveys the access permitted to the datum (i.e., either read-write or read-only). Then REF has no bearing on status, which is determined by another keyword, VAL.

So what does the REF (REFERENCE, PTR, etc.) mean? In ALGOL W and PL/I it is clearly an address value or variable (occupying some memory) of another object in memory. In MARY also, most REFs are addresses, but there is an implicit VAL REF for each identifier or literal which may not appear in memory. In ALGOL 68, we may say that each REF is a conceptual pointer which has a side effect of making variable the object referred to. How should conceptual pointers be mapped to physical pointers (i.e., addresses in memory) at run time? If one uses one physical pointer for each conceptual pointer (i.e., one for each REF in the mode), then integer variables will be twice as big in ALGOL 68 as in FORTRAN (clearly unacceptable). They will also be twice as slow when using their values. Also, large constants must then be copied instead of passed by reference, which may require an extra physical pointer. One could use as many physical pointers as REFs in the declarer, this often being one less the the number of REFs in the mode; this is efficient for simple variables, but not for "renamed" variables. If a "most efficient" map is used, then the programmer may have difficulty in knowing what that map is. Whereas a user of ALGOL 68 may be pacified by saying that the run-time organization is irrelevant to him as long as it works, a user of a MOL will not be. Therefore, if the ALGOL 68 model were to be used in PL370, some sub-optimal compromise would be required.

Like MARY, PL370 rejects the ALGOL 68 model of modes. That is, an integer variable is not a reference to an integer constant, but rather integers come in at least two kinds: constants and variables. In a sense, this is a retreat from ALGOL 68 back to ALGOL W, but both MARY and PL370 provide an even richer choice of modes than ALGOL 68, probably richer than any other language. In PL370, each object has a mode which includes a status. A status is either read-only (i.e., a constant or value) or read-write (i.e., a variable). The former is indicated in a declarer by CONS and the latter by VAR or by default. A reference is an address in memory constrained to point at objects of a given mode via a path of a given status (at least as restrictive as the object being pointed at). Thus a REF VAR INT may point at a VAR INT, but not at a CONS INT. In the case of fields of a structure, each field has a status and the structure as a whole also has a status. If the whole structure is read-only, then that status overrides those mentioned for each field. If the whole structure is read-write, then each field behaves according to its declared status, except that a read-only field may be changed by an assignment to the complete structure.

3.6 Packing of Data

Several languages have mechanisms whereby structures may exist in either a compact ("packed") format more suitable for transmission or in an expanded format more suitable for processing. For example, in COBOL, "MOVE CORRESPONDING" will copy all fields from one structure to the corresponding fields in another structure (two fields correspond if they have the same identifier). This feature will convert any fields that it can during the copy. (Actually, many COBOL structures are expanded for transput.) MARY, on the other hand, has an explicit PACKED keyword to tell the compiler to pack all the parts of the multiple value with this attribute as compactly as possible. One may pack or unpack by assignments from a PACKED (UNPACKED) object to an UNPACKED (PACKED) variable of the same mode.

Data is more useful in a packed format because it then takes less memory (or file space) to store it. In many files, with thousands or millions of data items, this is essential. Data is more useful in expanded format because then the CPU can process it directly.

What types of data might be packed?

- Integers in the range (-128 TO 127) or whole numbers in the range (0 TO 255) can be packed into one byte.
- 9 ASCII characters can be stored in a doubleword (8 bytes).
- Programmer-defined types can be stored using as few bits

as necessary, not some multiple of 8.

- Integers in the range (a TO b) can be mapped onto the range (0 TO b-a).
- Alignments required by machine instructions can be ignored.

In summary then, there are 3 aspects of expanding:

- representing all values "as is",
- inserting empty bits for alignment,
- inserting empty bytes for alignment.

A solution is to have a machine-dependent part of a mode declaration. The range shifting can be accomplished by an "EXCESS <compile-time-integer-expression>" part, meaning that the representation exceeds the desired value by that expression. The alignment can be controlled by an "ALIGNED <byte-boundary> <bit-boundary-option> " part; for example, "ALIGNED BYTE + 1 BIT", "UNALIGNED", "ALIGNED WORD+1" (i.e., WORD + 1 BYTE).

Move corresponding is not feasible unless different structure modes may have field selectors with the same identifiers. For those programmers whose style is to have a field selector indicate the mode (e.g. by the first few letters), move corresponding is useless. In any case, the programmer can build one of his own by defining the appropriate macros.

3.7 Storage Allocation

In most assembly languages, instructions and data are intermixed and appear in the object code in exactly the order presented. Some assembly languages give the programmer multiple location counters so that while all items using one counter appear as given, those using another may not appear until after everything using the first. Some let the programmer set the location counter, letting him overlay previously mentioned items or leave gaps.

In most higher-level languages, the programmer's control is less and indirect. A compiler may reorder variables to minimize waste space, move code from an internal procedure to somewhere outside, and reserve room for temporaries without ever reporting them. Variables are almost always kept separately from code, as are constants. The programmer does have some control over the number of objects that exist when he uses explicit allocation (e.g. generators in ALGOL 68, record designators in ALGOL W, and ALLOC in PL/I). PL370 must stride a middle road, not requiring any more work from the programmer than a high-level language, yet allowing him to control positioning as much as he wants to.

3.7.1 Location of Data

The basic control over location of data is the kind of usability desired; i.e., re-entrant, re-usable, or not re-usable. This, the programmer will specify for each module. Reordering of variables is an optimization that should not be done unless specifically requested, because it makes debugging much harder and many programmers have already carefully ordered their declarations.

Data cannot appear exactly where found in the code. Declarations are reached by flow of control in Algol-like languages, so, if each datum was located where declared, many extra jumps would be required in the code. The programmer can put data in a given order in a special section by using global records, but this is unnecessary in the normal case. When data and code are intermingled (non-re-entrant), the simplest safe action is to remember all of the data and reserve space for them after the nearest or the next unconditional branch. The data could also be gathered after each procedure, or after each global procedure, or in a separate section. Variables can be controlled independently of constants, although on the IBM 370 there is no advantage in putting the constants in a separate CSECT until IBM develops execute-only memory (i.e. fetching would be allowed only as an instruction, not by an instruction). The programmer should be allowed all these choices.

3.7.2 Dynamic Allocation

Most operating systems have one primitive for obtaining user storage from the system and one for returning storage. So this is probably a good model. Each module can specify a routine to be used to allocate dynamically and another to free. These will be invoked automatically when the user specifies re-entrant code. Such routines will all have the same required parameter and as many optional parameters as the programmer desires.

The user must also be able to allocate explicitly. One method is to consider all the available memqry as one area and let it be broken into sub-areas, which may in turn be split up, and so on. Another is for the user to write explicit generator routines, such as HEAP. These could be invoked as in ALGOL 68, and behave as the programmer defines. They can be easily used to create and maintain areas. They can also include garbage collection; since all allocation goes through the programmer's routine, each object may be allocated to include any extra information that the generator wants.

3.8 Register Management

3.8.1 Register Usage

In most languages, including PL360 and assembly languages, a programmer may store a value in a variable and subsequently store another value in the same variable without any warning. In many, however, he is warned if the second value has a mode different from the first. But since non-use of a value is usually an error, it might be preferrable to require the programmer to explicitly throw away values he has created but does not want to use. This is especially important for registers, since an unused value in a register is even more likely to indicate an error. There are five ways to access a register:

a) an otherwise ordinary declaration is located in a register rather than in memory.

b) as above, but a specific register is requested.

c) a specific register is used without any declaration to give its mode.

d) a part of a register is accessed as a field, considering the register as a structure.

e) a register is accessed as a side effect of another operation.

All five are useful and therefore should be included in PL370. Examples of uses of these five are:

a) to increase speed

b) a value is going to end up in a specific register anyway (e.g. a parameter) so why not put it there in the first place?

c) having performed an integer division, one may cheaply extract the remainder from the doubleword result

d) EDMK, TRT

e) after an access of type (d)

The compiler must carefully note which objects with different names actually affect each other. For example, in:

```
REG LONG REAL a,b; ¢ these are 16-byte reals
fr0 := 3.0;
```

the assignment <u>must</u> modify a or b, because there are only 32 bytes of floating-point registers. However the problem is not different for registers from that for memory.

3.8.2 Base Registers

3.8.2.1 Pointing at Data

IBMASM and PL360 show two different views of the world. In the former, data exists in organized structures. When a register points at such a structure, that register may be used explicitly (e.g. "DFIELD-DNAME(REG)") or implicitly after a "USING" statement. Note tht only one "record" may be used at any one time with any given mapping, but any number of records may be used by giving explicit pointers and offsets. IBMASM also allows the data definition to be used like a lorgnette, giving a particular view of an area. PL360, on the other hand, seems to reflect a view of the data mapping as being fixed to the register, rather than to the storage. In IBMASM, the strong link is memory <-> map and the weak link is register <-> map . In PL360, the reverse is true. Personally, I believe the IBMASM view to be more realistic and furthermore it matches the ALGOL 68 model better; in ALGOL 68, memory binding is fixed at generation time, but pointer binding is not fixed until the actual selection. Considering that the vast majority of IBMASM programmers choose to access records by USING a base register for them (and not mentioning it again), PL370 should include a similar mechanism. However, the ALGOL 68 model has the ability to access other records much more conveniently than IBMASM (when one record of that type has been opened), and this should not be lost.

3.8.2.2 Pointing at Code

Again, in IBMASM, the binding of code base registers is much more flexible than in PL360. In particular, the register may be changed part way through a procedure, and in fact, almost all IBMASM programmers take advantage of this (look at a standard entry macro [21]). PL360's restriction to the same register throughout is unacceptable and easy to overcome.

3.8.2.3 Large Sections

On the IBM 370, handling sections longer than 4 kilobytes is always a problem. PL360 avoids it by forcing the programmer to break his procedure into segments each of 4KB or less. IBMASM provides two additional methods. The first is to have more than one register for the code, thus allowing lengths of 4KB times the number of dedicated registers. Perhaps more important, there is no requirement that one point at the beginning of the procedure, only that the area "covered" by the register be sufficient. However, this method requires some run-time management, which, in IBMASM, is supplied by the programmer. The programmer may :

- Use an extra base register.
- Break his procedure into 2 pieces.
- Draw a line about 4000 bytes from the beginning of the procedure and base code from there when past that line. Crossing that line then requires modifying the base register. Constants may be duplicated or pointed at via a full address from across the line.
- Make the constants addressable at all times (e.g. located at the beginning of the proc) and use long jumps in the latter part of the procedure.

PL370 should allow all these options, but it would be nice if one of the last two could be automated (the last one is easier to do).

3.8.3 Register Names

What should the registers be called? While this is not a crucial question (anyone can make synonyms to suit his taste), the answer may be important to the popularity of PL370.

(a) general registers

modes held: INT, BITS, WHOLE, BIT, BOOL, CHAR, ADDRESS, SHORT WHOLE, SHORT INT, CHARS(2), CHARS(3), CHARS(4)

(1) r0, r1, ..., r15 (like PL360)
(2) gr0, gr1, ..., gr15 (like MTS)
(3) r0, r1, ..., r9, ra, rb, ..., rf (used in OS)
(4) regs(0), regs(1), ..., regs(15)
(5) reg(0), reg(1), ..., reg(15)
(6) gregs(0), gregs(1), ..., gregs(15)
(7) greg(0), greg(1), ..., greg(15)

(b) floating-point registers(short)

modes held: SHORT REAL

(1) f0, f2, f4, f6 (like PL360)
(2) fr0, fr2, fr4, fr6 (like MTS)
(3) f0, f1, f2, f3
(4) fr0, fr1, fr2, fr3
(5) fregs(0), fregs(2), fregs(4), fregs(6)
(6) freg(0), freg(2), freg(4), freg(6)
(7) fregs(0), fregs(1), fregs(2), fregs(3)
(8) freg(0), freg(1), freg(2), freg(3)
(9)-(16) as (1)-(8), but preceeded by an 's' (for short)

(c) control registers

modes held: depends on register

(1) c0, c1, ..., c15
(2) cr0, cr1, ..., cr15
(3) c0, c1, ..., c9, ca, cb, ..., cf

(4) cregs(0), cregs(1), ..., cregs(15)
(5) creg(0), creg(1), ..., creg(15)

(d) floating-point registers(long)

modes held: REAL

(1) f0, f2, f4, f6 (since mode=REAL)
(2) fr0, fr2, fr4, fr6
(3) f0, f1, f2, f3
(4) fr0, fr1, fr2, fr3
(4) fregs(0), fregs(2), fregs(4), fregs(6)
(5) freg(0), freg(2), freg(4), freg(6)
(6) fregs(0), fregs(1), fregs(2), fregs(3)
(7) freg(0), freg(1), freg(2), freg(3)
(8) f01, f23, f45, f67 (like PL360)
(9) fr01, fr23, fr45, fr67
(10) fregs(0:1), fregs(2:3), fregs(4:5), fregs(6:7)
(11) freg(0:1), freg(2:3), freg(4:5), freg(6:7)
(12)-(18) as (1)-(7), but preceeded by a 'd' (for double)

(e) floating-point registers(extended)

modes held: LONG REAL

(1) lf0, lf4 (since mode=LONG REAL)
(2) lfr0, lfr4
(3) lf0, lf1
(4) lfr0, lfr1
(4) lfregs(0), lfregs(4)
(5) lfreg(0), lfreg(4)
(6) lfregs(0), lfregs(1)
(7) lfreg(0), lfreg(1)
(8) f0123, f4567 (PL360 style)
(9) fr0123, fr4567
(10) fregs(0:2), fregs(4:6)
(11) freg(0:2), freg(4:6)
(12) fregs(0:3), fregs(4:7)
(13) freg(0:3), freg(4:7)
(14)-(20) as (1)-(7), but preceeded by a 'q' (for quadruple) rather than an 'l'.

(f) general register pairs

modes held: LONG INT, LONG WHOLE, LONG BITS, CHARS(8)

(1) dr0, dr2, ..., dr14
(2) d0, d2, ..., d14
(3) dgr0, dgr2, ..., dgr14
(4) gdr0, gdr2, ..., gdr14
(5) gd0, gd2, ..., gd14
(6) dg0, dg2, ..., dg14
(7) dr0, dr2, ..., dr8, dra, drc, dre
(8) dr0, dr1, ..., dr7

(9) d0, d1, ..., d7
(10) dgr0, dgr1, ..., dgr7
(11) gdr0, gdr1, ..., gdr7
(12) gd0, gd1, ..., gd7
(13) dg0, dg1, ..., dg7
(14) dregs(0), dregs(2), ..., dregs(14)
(15) dgregs(0), dgregs(2), ..., dgregs(14)
(16) gdregs(0), gdregs(2), ..., gdregs(14)
(17) dregs(0), dregs(1), ..., dregs(7)
(18) dgregs(0), dgregs(1), ..., dgregs(7)
(19) gdregs(0), gdregs(1), ..., gdregs(7)
(20) dreg(0), dreg(2), ..., dreg(14)
(21) dgreg(0), dgreg(2), ..., dgreg(14)
(22) gdreg(0), gdreg(2), ..., gdreg(14)
(23) dreg(0), dreg(1), ..., dreg(7)
(24) dgreg(0), dgreg(1), ..., dgreg(7)
(25) gdreg(0), gdreg(1), ..., gdreg(7)
(26) regs(0:1), regs(2:3), ..., regs(14:15)
(27) gregs(0:1), gregs(2:3), ..., gregs(14:15)
(28) reg(0:1), reg(2:3), ..., reg(14:15)
(29) greg(0:1), greg(2:3), ..., greg(14:15)

3.9 Scope

Since this is an ALGOL-like language, the basic scope rules are derived from block structure. That is, in general, an inner block may reference any object declared in a block containing it, but an outer block may not reference any objects declared in blocks it contains. It would be quite useful to have both own (static) and ordinary (automatic) variables. References are a little more complicated. In ALGOL 68, the mode REF AMODE is really a union whose meaning changes in every range. This is because a REF AMODE object refers to an AMODE object of any scope greater than or equal to that of the REF. This is effectively UNION(REF HEAP-AMODE,REF global-AMODE,REF nest1-AMODE,REF nest2-AMODE,...,REF LOC-AMODE), and in fact, a proper ALGOL 68 system must break open the union on almost any assignment of a REF without dereferencing. Four possible ways to handle this problem are:

(1) Ignore the problem and never check for errors. Allow all the flexibility of ALGOL 68, but force the programmer to explicitly insert code to check for errors. This is the method used by ALGOL68C [8].

(2) Allow full ALGOL 68 flexibility and use a proper scope check. This can get pretty expensive at run-time.

(3) Disallow any pointer assignment unless guaranteed safe at compile-time. This is cheap to implement in the compiler and eliminates all need for run-time checking, but does restrict the language, forcing many objects from the stack to the heap. At this level, that may not be much of a loss.

(4) As (3), but extend the language to allow declarations of the form:

```
    :p:   BEGIN
       INT ip;
       ...
          BEGIN
          INT i3;
          REF INT p1;    ¢ as flexible as ALGOL 68
          p1 := @ip; ¢ legal
          p1 := @i3; ¢ legal
          REF LOC(p) INT p2;
          p2 := @ip; ¢ legal
          p2 := @i3; ¢ illegal
          p1 := p2; ¢ legal
          p2 := p1; ¢ illegal
          REF HEAP INT p3;
          p3 := HEAP INT; ¢ legal
          p3 := @i3; ¢ illegal
```

```
p3 := @ip; ¢ illegal
p3 := p1; ¢ illegal
p3 := p2; ¢ illegal
p1 := p3; ¢ legal
p2 := p3; ¢ legal
```

If (3) or (4) be chosen, then many variables will be declared more global in scope in order that they may be shared. In that case, one or more of the protection schemes below may be required.

Certain deviations from strict nesting scope rules may be useful for efficiency or protection.

Two scopes at the same level could share *own* objects (to the exclusion of their common container) by mutual co-operation. This can be simulated by the PERMIT below, but that involves trusting the containing procedure.

One can compare data to files. Just as advanced operating systems allow various permissions to different users of a file, a language could allow various permissions to different users (read ranges or blocks or procedures) or an object. This could be implemented by a PERMIT command much like that in MTS [35]. i.e.:

```
PERMIT <identifier-list> [  R  |  RW  |  NONE  ]
<range-name-list> ;
```

This could be implemented at run-time (easier to understand but expensive) or at compile-time. As a compile-time tool, this is much clumsier, but the difference in cost (none versus much) is very great.

One could restrict the "natural" flow of scope into the next level by having a block explicitly name those non-local objects which it might use. Since this acts as a filter, it can be used both by the container to restrict what is passed on, and by a block to guarantee it does not ruin its environment. The default import list could be made all or none. The latter is safer and more efficient, whereas the former is more like ALGOL 68 and may save some typing. The import list is an important consideration when efficiency is concerned because if all outer scopes must be available in case needed, then, in general, there is a certain amount of overhead at each scope entry and exit for addressability.

In a true stack scheme, it should be possible to allocate objects in the frame of the calling procedure. This is because there is nothing above that frame until the new procedure creates its own, but it could put some objects on the old frame first. This capability is built into most such schemes for the return of values. It would be a useful general facility, but would not be free, because the local frame of a procedure would no longer be static in size. This means that a stack-top or frame-size variable would have to be kept around and updated at run-time.

3.10 Addressability

This section is very much tied up with register management (q.v.) and scope. Depending on register usages, access to non-local storage can be made fast (limited number of ranges), slow (unlimited number of ranges, but getting even slower as ranges get more non-local), or impossible (as in FORTRAN). Global references can be handled as extreme non-local (as in ALGOL W) or as a special case (as in ALGOL68C). The usual way to address an object is to give the compiler a register to use for this purpose, like a USING in IBMASM. Another way is to dereference a pointer.

To address code, there must be at least one register given to the compiler. This can be done with a statement of the form:

```
BASE CODE ON <address-expression>
```

This will cause the compiler to use the specified address expression for addressing all the code with the current procedure, except for any contained procedures which have their own BASE statements. It would be useful if one could base a structure on any address expression, not just a register as in IBMASM, e.g.:

```
register
register + offset
register + register
register + register + offset
actual address
```

Another convenience would be to have the compiler generate code to perform the copy when a BASE statement indicates that the base register is being switched from one register to another. Note that since BASE CODE statements usually appear in an ENTER macro, the usual program can ignore this item altogether. Large sections of code can be handled by explicitly naming extra base registers, e.g.:

```
      BASE  CODE  ON  r3, r4
```

They can also be handled by the compiler automatically taking necessary actions (see Section 3.8.2.3).

There are 4 different ways to address data.

(1) The basic data blocks are specified as for code, when located separately. VARS may be specified when the variables are located separately from the code, and CONS may be specified only when the constants are located separately from both the code and the variables:

```
      BASE  VARS  ON  <address-expression>
      BASE  CONS  ON  <address-expression>
```

(2) Global records may addressed in a similar manner. Each one needs one base register, since the usual rule is that separate CSECTs are loaded independently:

```
      BASE  <record name>  ON  <address-expression>
```

(3) If ST is a structed mode, and register r contains a value of mode REF ST or PTR ST, then any field of ST may be accessed by:

```
      <field name>  OF  r
```

(4) If ST is a structed mode, and register r contains a value of mode REF ST or PTR ST, and if the particular object pointed to by r is being used frequently, then it may be more convenient to "open" the structure:

```
OPEN r;
f := 3;
g := 3;
h := 3.0;
       is equivalent to:
f OF r := 3;
g OF r := 3;
h OF r := 3.0;
```

This "open"ing lasts until the end of the clause in which the structure was opened or until r is assigned a new value, whichever comes first. The open statement is a copy of IBMASM's USING statement, and is also similar to Pascal's with statement.

3.11 Multiple Values

In ALGOL 68, a row is more than just a sequence of elements - it also includes a descriptor. There is no mode which means just the elements (which is all you get in many languages). In PL370, it is required that both rows with desciptors and rows without descriptors be available; therefore the primitive ARRAY is the more primitive construct - just the elements. A ROW is then the descriptor for an ARRAY.

Since one aim is for compatability with the language being replaced, and since array formats vary so widely, there must be a compromise. Two schemes have been considered:

(1) Let the programmer define the " [ ] " operator; or use the standard definitions when applicable. This allows him to store an array in any manner he wishes, such as row-major order or column-major order. He may check subscripts if he wishes, or do so only for certain modes. Unfortunately, or fortunately, depending on your bias, this turns each array into a procedure delivering an element and is basically the same as what Bliss has done.

(2) Provide a less flexible construct that cannot build the general procedures of the first scheme, but requires no "programming" and covers all reasonable cases. This can be used if you agree that:

- There is no reasonable way to store an array other than row-major order or column-major order. Scheme (1) can use a sparse representation instead of the usual full representation without requiring any change in the program outside the definition of subscripting.
- Subscript checking should be either complete or left out completely (for any one mode).
- A default lower bound makes sense, but a default upper bound does not.
- There are really no other important parameters to element-storing mechanisms.

This scheme generalizes ARRAY by allowing different ARRAYTYPEs bound by three characteristics:

```
( { ROW-MAJOR | COLUMN-MAJOR },
  { CHECK | NOCHECK | },
  default lower bound )
```

Examples:

```
ARRAYTYPE  NICE_ARRAY = ( ROW-MAJOR, CHECK, 1 );
NICE_ARRAY ( 5 ) INT f;
NICE_ARRAY ( 5:6, -9:5 ) COMPL x;
```

Note that if a NICE_ARRAY is passed as an actual parameter, the mode matching requires that the bounds match also.

Now that the elements have been described, one must design a format for the descriptor. One can define descriptors in terms of STRUCTs. Suppose that there is a compile-time procedure 'dim' which for any array mode returns the number of dimensions in that mode. One can then define a ROW by:

```
DESCRIPTOR  ROW (X) = STRUCT (
      REF X  p;
      ARRAY(1:dim(X))  lower bounds,
                       upper bounds
                              );
```

A more comprehensive solution to rows is to have the user define complete modules in the sense of Zilles, Liskov, etc. [11]. A module would describe how to represent a descriptor, how to allocate one, how to assign them, and how to use them to reference an element of the array. This is even more comprehensive than the previous scheme for defining subscripting, because the programmer has control over copying as well.

3.12 Linkage Conventions

This topic concerns the implementation of procedure calling. Most implementations pick one method and stick to it. Unfortunately, most of them pick a different method. PL360 and IBMASM avoid this problem and leave it to the programmer, who must define procedure call explicitly every time. In practice, the IBMASM programmer can make order out of chaos by judicious use of macros, usually one each of CALL, ENTER, and EXIT for each linkage convention. Even so, it is very easy to make errors, and many errors occur at these interfaces between procedures. Therefore, PL370 seeks to give the programmer even more control over his linkage without restricting valid flexibility.

One important consideration of a linkage convention is restoration of the "state of the world" to what it was before the start of the call, excepting, of course, those changes that the procedure was called to effect. Responsibilities for restoration are usually divided between the caller and the callee, but the programmer must be careful that he has covered as much as he can between them. A compiler should assist the programmer with this. However the list of things to restore may be outrageously long, or an operating system may be deficient and unable to restore something, once changed. For example, see the table describing MTS.

ITEM	CAN GET VALUE?	CAN CHANGE VALUE?	CAN RESTORE VALUE?
general registers	yes	yes	yes
floating-point registers	yes	yes	yes
program mask	yes	yes	yes
condition code	yes	yes	yes
instruction address	yes	yes	yes
current program trap	yes	yes	yes
current attention trap	yes	yes	yes
current time trap	no	yes	no
current prefix char	yes	yes	yes
control chars (#,>,-,:,etc)	yes	yes	yes
accounting statistics	yes	no	no
other "User INFOrmation" [21]	yes	some	some
I/O assignments	yes	yes	no
local time limit	yes	no	no
global settings ($SET)	yes	yes	yes
current active file	no	yes	no
current source file	no	yes	no
current sink file	no	yes	no
set of valid pseudo-names	no	yes	no
for each tape - using labels?	yes	yes	yes
for each tape - format	yes	yes	yes
current position of a tape	yes	yes	yes
for each tape - other info	some	yes	some

Figure 3. States of the World under MTS [21]

Some of the things commonly considered part of a linkage convention are:

- What registers contain information at procedure entry?
- What are their modes and values?
- What registers are saved just after entry?
- What else is done at entry time?
- What registers contain information just after procedure exit?

- What are their modes and values?
- What registers are restored just before exit?
- What else is done at exit time?
- How does one call?
- What parameters are passed in registers?
- How are parameters passed in memory?
- Does the parameter list contain values or addresses?
- How are optional parameters handled?
- What information must be saved and restored by the caller?
- How does a procedure obtain and release local storage?
- What kind of save areas are maintained?

It is very difficult to succinctly characterize a linkage convention. What is possible, is to provide a framework in which the compiler and the programmer can co-operate to the benefit of the programmer. By means of declarations of the important states of the world at the procedure call boundary, the compiler can check that at least the modes match. The three main parts which the user must define by himself are entering, calling, and exiting. It would be nice to provide a generator that could be given some parameters and work in all cases, but conventions are so diverse that any generator would be full of kludges to cover the exceptions in the more common conventions. The only all-purpose way is complete specification by the programmer.

A linkage convention definition will specify:

- what registers contain information on entry and what mode of information, (special "modes" such as ENTRY, RETURN, and PLIST will be allowed here),
- what registers contain information on exit and what mode of information,
- what registers contain junk on exit,
- (by omission) what registers will have been restored on exit,
- how to call,
- how to enter,
- how to exit,
- how to save and link,
- how to unlink and restore, and
- how to return a value (for a function (see Section 3.18)).

Once linkage conventions have been defined, each procedure is associated with one convention. Clearly, IBM's S-type and R-type conventions must be pre-defined so that everybody can use them.

The compiler can then ensure that at the time of a call, all input registers have, in fact, been given acceptable values. It also knows which registers must be stored in memory (or lost) across the call, and which can be left alone. It also can maintain its knowledge of the current modes of registers.

In order to help the programmer define entry, call, and exit, the compiler will provide some primitives to give him some information. Some obvious ones are the number of parameters expected, their types, how many actual parameters there are, and the mode of the result (if any) that the procedure is supposed to produce.

3.13 Operators

ALGOL 68 introduced user definable operators. They introduce no additional capablilities over procedures, but can enhance readability tremendously when used correctly. Therefore, PL370 should include at least the ALGOL 68 concept.

Since operators are usually used for short, frequent code sequences, it might be useful to distinguish between operators and procedures by normally generating in-line code for the operators, and out-of-line code for the procedures.

3.14 Exception Handling

The programmer needs some way to control what happens after a program interrupt or after an error that has been caught by software. He has some control, because he checks for some errors himself, and because he can easily call the appropriate system routines to set up an interrupt handler. However, these global error handlers have much overhead, are difficult to write, and are clumsy. A small routine to catch a local possible error is easier to write. The point is made relevant by the existence of a "branch on program interrupt" (BPI) feature in MTS [21]. One can set a local trap for a possible error in the immediately preceeding instruction. Although this facility is not available in the much more prevalent OS, a user could add it via a global handler without much difficulty - if he thought of it. In practice, BPI allows intelligent action after an interrupt almost every time, when a global error routine would be floundering around for an excuse. It is, therefore, important that PL370 be able to support this feature. Because it applies only to a single instruction, and some PL370 primitives may generate several, the IBMASM technique of mentioning it after the fallible instruction won't work.

At a little higher level, the compiler may be able to generate better code if it knows what the program mask will be at run-time. For instance, there is no point in generating branch-on-overflow instructions when the mask will be set for overflow to generate an interrupt. Some features like the PL/I on unit and the PL/I condition prefix may be useful. The aim is to connect a recovery routine with a range of source text and for a given class of errors.

It can be appended to the statement covered (as in COBOL), which matches the BPI usage in IBMASM closely, or it can be turned ON and OFF (at compile-time) like a PL/I on unit. The second is certainly easier to compile, because the class of errors to check for is known before the clause is read.

Most languages that do check for subscript errors provide no method for recovery from that error. One possible method, mentioned in the section about case clauses, is to provide an "OUT" clause for the subcript, which would be elaborated if the subcript(s) were not correct; the OUT clause would be of the same mode as an array element and would be the value of the subscripted obect.

3.15 Basic Actions

3.15.1 Introduction

This section concerns the simplest statements and expressions out of which programs are composed. All the more complex control structures eventually invoke these "basic actions". For example, in ALGOL 60, the basic actions are:

assignment statement
procedure call
dummy statement
go to

In FORTRAN, there are six categories of basic actions:

assignment statement (including ASSIGN)
CALL
CONTINUE
GO TO
READ
WRITE

In a machine language, however, there are many dozens of instructions, each one of which is a basic action of that machine language. Many fit into the categories named above, but some don't. PL360 neatly put 53[1] instructions into the first four categories and defined functions as a method of acessing any other single instruction. A function call looks like a procedure call, but the function is defined by a single machine instruction in an assembler-like format. Another method used by PL360 is operators, the destination of whose results is restricted.

The term "basic actions" can be interpreted in two ways. First, how can the machine primitives be represented (requested) in a PL370 program; this is equivalent to asking how to decompile an arbitrary machine language program. Second, what are the primitive actions of PL370, those that can be done without requiring the programmer to write more than a simple expression, even if they require several machine instructions? It is the first interpretation which is taken here.

---

[1] 51 for assignment and arithmetic (see Table 4 in Section 4) + BC + BAL

3.15.2 Representation of Individual Instructions

Most of the time, one does not care whether a source construct is translated into one instruction or several (aside from efficiency considerations). However, the execute instruction has as its operand, exactly 1 instruction. This effectively makes instruction a data type, which needs some way of denoting objects of that type. One possibility is to use PL360's Assembler-like functions. Another is to provide an Algol-style representation for each operation.

An aim of PL370 is to account for most of the IBM 370 instructions without resorting to functions. Two techniques used are better data descriptions and the "op-and-becomes" format. We start off with a simple base, using the following modes:

PL370 mode	PL360 mode	IBMASM type code
INT	INTEGER	'F'
SHORT INT	SHORT INTEGER	'H'
REAL	LONG REAL	'D'
SHORT REAL	REAL	'E'
WHOLE	LOGICAL	

CHAR	BYTE	'C'

Many operators are easily accounted for using the basic patterns " a := b " and " a *op*:= b " (i.e., " a := a *op* b "). That is, the contents of "a" are replaced by the value of "b" or "a *op* b". Although the terms "INT reg", "WHOLE reg", "CHAR reg", and "SHORT INT reg" appear below, they must all end up as general registers; however, in this section it is assumed that a compiler can know the current mode of every register. For a source register, INT will accept INT, WHOLE, or SHORT INT. For a destination register, the operation may change the current mode to that specified. For instance:

```
SHORT INT i = 2;
INT j = 100000;
r1 := i;          ¢     r1 is now SHORT INT
r1 +:= j;         ¢     r1 is now INT
```

The notation used below is:

<AMODE reg> - a register whose current mode is AMODE

<AMODE mem> - an AMODE object in memory

<AMODE literal> - an AMODE denotation

<addr> - an address expression

- <WHOLE literal> *or*
- <WHOLE reg> + <WHOLE literal>

(1) *copy*

L	<INT reg> := <INT mem>
LD	<REAL reg> := <REAL mem>
LDR	<REAL reg> := <REAL reg>
LE	<SHORT REAL reg> := <SHORT REAL mem>
LER	<SHORT REAL reg> := <SHORT REAL reg>
LH	<SHORT INT reg> := <SHORT INT mem>

LR	<INT reg> := <INT reg>
MVI	<CHAR mem> := <CHAR literal>
ST	<INT mem> := <INT reg>
STD	<REAL mem> := <REAL reg>
STE	<SHORT REAL mem> := <SHORT REAL reg>
STH	<SHORT INT mem> := <SHORT INT reg>

(2) add

A	<INT reg> +:= <INT mem>
AD	<REAL reg> +:= <REAL mem>
ADR	<REAL reg> +:= <REAL reg>
AE	<SHORT REAL reg> +:= <SHORT REAL mem>
AER	<SHORT REAL reg> +:= <SHORT REAL reg>
AH	<INT reg> +:= <SHORT INT mem>
AL	<WHOLE reg> +:= <WHOLE mem>
ALR	<WHOLE reg> +:= <WHOLE reg>
AR	<INT reg> +:= <INT reg>

(3) subtract

S	<INT reg> -:= <INT mem>
SD	<REAL reg> -:= <REAL mem>
SDR	<REAL reg> -:= <REAL reg>
SE	<SHORT REAL reg> -:= <SHORT REAL mem>
SER	<SHORT REAL reg> -:= <SHORT REAL reg>
SH	<INT reg> -:= <SHORT INT mem>
SL	<WHOLE reg> -:= <WHOLE mem>
SLR	<WHOLE reg> -:= <WHOLE reg>
SR	<INT reg> -:= <INT reg>

(4) multiply

MD	<REAL reg> *:= <REAL mem>
MDR	<REAL reg> *:= <REAL reg>
MH	<INT reg> *:= <SHORT INT mem>

(5) divide

DD	<REAL reg> /:= <REAL mem>
DDR	<REAL reg> /:= <REAL reg>
DE	<SHORT REAL reg> /:= <SHORT REAL mem>
DER	<SHORT REAL reg> /:= <SHORT REAL reg>
HDR	<REAL reg > /:= 2
HER	<SHORT REAL reg > /:= 2

(6) *and*

N	<WHOLE reg>	AND:=	<WHOLE mem>
NI	<CHAR mem>	AND:=	<CHAR literal>
NR	<WHOLE reg>	AND:=	<WHOLE reg>

(7) *inclusive or*

O	<WHOLE reg>	OR:=	<WHOLE mem>
OI	<CHAR mem>	OR:=	<CHAR literal>
OR	<WHOLE reg>	OR:=	<WHOLE reg>

(8) *exclusive or*

X	<WHOLE reg>	XOR:=	<WHOLE mem>
XI	<CHAR mem>	XOR:=	<CHAR literal>
XR	<WHOLE reg>	XOR:=	<WHOLE reg>

(9) *shifts*

SLA	<INT reg>	SHL:=	<addr>
SLL	<WHOLE reg>	SHL:=	<addr>
SRA	<INT reg>	SHR:=	<addr>
SRL	<WHOLE reg>	SHR:=	<addr>

Note that AL and SL have been changed from PL360, where the operator is different ("++") and the modes INTEGER and LOGICAL are not distinguished.

Another basic pattern is the monadic operator. It appears in two forms:

(i) a := *op* b

(ii) *op* a

PL360 uses the operator symbols *neg*, *abs*, and *neg abs*. Another operator which fits this paradigm is copy-and-set-condition-code, or *test* for short. Rounding from one length of *real* to a shorter one can be done by assignment, but Algol tradition holds that the programmer must explicitly acknowledge that he is throwing away precision, by using a monadic operator such as *short*. It is also tempting to shorten (i) even further to

(iii) a *op*:= b

See the example shown below for LTR.

(10) *operators*

BAL	CALL <procedure-identifier>
BALR	CALL <REF PROCMODE reg>
LCDR	<REAL reg> := NEG <REAL reg>
LCER	<SHORT REAL reg> := NEG <SHORT REAL reg>
LCR	<INT reg> := NEG <INT reg>
LNDR	<REAL reg> := NEG_ABS <REAL reg>
LNER	<SHORT REAL reg> := NEG_ABS <SHORT REAL reg>
LNR	<INT reg> := NEG_ABS <INT reg>
LPDR	<REAL reg> := ABS <REAL reg>
LPER	<SHORT REAL reg> := ABS <SHORT REAL reg>
LPR	<INT reg> := ABS <INT reg>
LRER	<SHORT REAL REG> := SHORT <REAL REG>
LTDR	<REAL reg> := TEST <REAL reg>
LTER	<SHORT REAL reg> := TEST <SHORT REAL reg>
LTR	<INT reg> := TEST <INT reg>
-	<INT reg> TEST:= <INT reg>

This accounts for 67 out of 185[1] instructions, leaving 118:

AP, AU, AUR, AW, AWR, AXR, BALR (non-branching), BC, BCR, BCT, BCTR, BCTR(non-branching), BXH, BXLE, C, CD, CDR, CDS, CE, CER, CH, CL, CLC, CLCL, CLI, CLM, CLR, CLRIO, CP, CR, CS, CVB, CVD, diagnose, D, DP, DR, ED, EDMK, EX, HDV, HIO, IC, ICM, IPK, ISK, LA, LCTL, LM, LPSW, LRA, LRDR, M, MC, ME, MER, MP, MR, MVC, MVCL, MVN, MVO, MVZ, MXD, MXDR, MXR, NC, OC, PACK, PTLB, RDD, RRB, SCK, SCKC, SIGP, SIO, SIOF, SLDA,

---

[1] There are 183 decribed in [6], but I am counting non-branching BALR and BCTR separately.

SLDL, SP, SPT, SPX, SPM, SPKA, SRDA, SRDL, SRP, SSK, SSM, STAP, STC, STCK, STCKC, STCM, STCTL, STIDC, STIDP, STM, STNSM, STOSM, STPT, STPX, SU, SUR, SVC, SXR, SW, SWR, TCH, TIO, TM, TR, TRT, TS, UNPK, WRD, XC, ZAP

3.15.3 Covering the Rest of the Instructions

There are five techniques for achieving complete coverage of the instruction set. These are:

(1) Invent new modes.

(2) Invent predefined variables (as PL360 does for registers).

(3) Invent new operators.

(4) Invent new syntax.

(5) Invent a clean procedural interface. This is pretty close to giving up and using a function.

Most of the multiplication instructions have been omitted because of a problem in semantics: the interpretation given to "a *:= b" by ALGOL 68 is that the multiplicand is "a", but these instructions assign to a register the product of one half of the register multiplied by another object the same size as the multiplicand. Consider the instruction (given in IBMASM format) MR 4,7: the multiplier is r7 (as expected), but the multiplicand is r5, while the product is both of r4 and r5. A more realistic representation might be:

```
dr4 := r5 * r7
```

This form reads better than the alternative:

```
dr4 *:= r7
```

However, it appears to have 3 independent operands, when, in fact the second operand must be the right half of the first operand. A more convenient story (for the program writer) would be to have a mode PAIR = STRUCT(INT even,odd), and to say that "*:=" can take a PAIR first operand and convert it into a LONG INT, but this contradicts the ALGOL 68 model of an operator, which would allow this subterfuge only by declaring *:= as an OP(UNION(PAIR,LONG INT),CONS INT). It is hard to choose between the two forms, but readability is probably more important than writeability; also, the longer form can exist with 3 independent operands and require two instructions in general.

A similar problem of semantics exists for integer divide, which replaces the dividend by two smaller integers in the place of the larger one. The easier-to-write choice is:

```
/*     The dividend is nicely prepared
      in a double register or in memory    */
<dividend> DIV:= <divisor>;
/*     The mode of the <dividend> is now
       STRUCT(INT remainder,quotient)
       and not LONG INT       */
```

The more difficult choice offers some safety advantages:

```
DIVIDE <dividend> BY <divisor> GIVING <tag1>,<tag2>
```

These <tag>s would be very localized; the <dividend> would be inaccessible under its previous <tag>, but the remainder and quotient fields would be accessible under <tag1> and <tag2>, until another value was moved into the variable.

3.15.4 New Modes

All of the modes previously presented can appear in both memory and registers. Of the modes presented below, only LONG INT, LONG WHOLE, CHARS(4), CHARS(8), LONG REAL and any REF mode can appear in registers.

- The IBM 360 and 370 series both have decimal hardware in addition to the binary that most procedural languages stick to. Unlike binary integers which must be either 2 bytes or 4 bytes, decimal integers may be any number of bytes from 1 to 16; therefore, it is necessary to have 16 decimal modes compared to 2 binary, and LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG LONG DECIMAL is out of the question. A more reasonable notation is DECIMAL(n), where $1 \le n \le 16$, and n is known at compile-time.
- The machine also supports another version of decimal numbers more suited for display than arithmetic, but still signed; these are similarly of any size from 1 to 16 and may be denoted ZONED(n), where $1 \le n \le 16$.
- The ALGOL 68 mode [ ]CHAR seems particularly appropriate on IBM hardware, which is character oriented. It will be included, but for convenience may be spelt CHARS(n). Here, 'n' may be in the range $1 \le n < 2**24$, but the operators

given below are defined only for $(1 \leq n \leq 256)$.

- A fourth mode, available only on larger machines is LONG REAL (called extended precision by IBM), which is twice as big as REAL.
- PL370 will use full pointer typing as in ALGOL 68 (see section 3.7), whereas PL360 uses INTEGER for addresses. Thus for any mode AMODE, there is a mode REF AMODE, which is a fullword pointer to an AMODE. The enreferencing operator is "@" - the same as in PL360.
- Since M and MR produce 64-bit signed binary products and D and DR use 64-bit signed binary dividends, we need LONG INT.
- Since the time of day clock is an unsigned 64-bit value, we need LONG WHOLE.
- Various pieces of the hardware use unsigned halfword integers - SHORT WHOLE. Many hardware-defined structures contain numbers of unusual sizes. The easiest way to support this is to allow INTs and WHOLEs of arbitrary sizes; of course, these must be copied into a standard sized variable before they can be used in arithmetic, but some can be compared as is. In keeping with the pattern used by DECIMAL, INT(n) means an INT n bytes long; INT(0,n) means an INT n bits long. WHOLE(n) and WHOLE(0,n) are defined similarly. Another necessary mode is BIT - one bit. Arrays of BITs can be called BITS(n) or ARRAY(n)BITS; BYTE means BITS(8) aligned in one byte.
- The psw and other special registers in the machine can

be supported by a variety of structured modes; these structures are described in detail in [6]. We need the mode PSW (some of its submodes are PROGRAM_MASK = BITS(4), PROTECTION_KEY, and CC = WHOLE(0,2)), STORAGE_KEY, PREFIX, and the mode CPUID = STRUCT(CHAR version code, CHARS(3) identification number, CHARS(2) model number, SHORT WHOLE max mcel length).

- It is possible to use a register to hold just one character and to ignore the high-order three bytes. In this case, we will say that the mode of the register is CHAR.
- It is possible to have a 3-byte address in a word with other information in the first byte. This simple structure can be called "PTR WITH ...", where "..." is any one-byte mode; e.g. PTR WITH (CHAR letter).
- Since a general register can hold so many different modes, and since it is often difficult to know which one it has at any given time, it is useful to have a name for the union of all possible modes which can be stored in a general register. This union is called GENRL.
- <u>Summary</u> - modes added : BIT, BITS(n), BYTE, CHARS(n), CPUID, DECIMAL(n), GENRL, INT(n), INT(0,n), LONG INT, LONG REAL, LONG WHOLE, PREFIX, PROGRAM_MASK, PROTECTION_KEY, PSW, PTR WITH ..., REF ..., STORAGE_KEY, SYSTEM_MASK, WHOLE(n), WHOLE(0,n) and ZONED(n).

These new modes cover the following 23 instructions from the long list given above:

```
AP       <DECIMAL(m) mem>  +:=  <DECIMAL(n) mem>
AXR      <LONG REAL reg>  +:=  <LONG REAL reg>
```

CVB	<INT reg> := <DECIMAL(8) mem>
CVD	<DECIMAL(8) mem> := <INT reg>
IC	<CHAR reg> := <CHAR mem>
LA	<REF AMODE reg> := @ (AMODE mem>
LRDR	<REAL reg> := SHORT <LONG REAL reg>
MP	<DECIMAL(m) mem> *:= <DECIMAL(n) mem>
MVC	<CHARS(n) mem> := <CHARS(n) mem>
MXR	<LONG REAL reg> *:= <LONG REAL reg>
NC	<CHARS(n) mem> AND:= <CHARS(n) mem>
OC	<CHARS(n) mem> OR:= <CHARS(n) mem>
PACK	<DECIMAL(m) mem> := <ZONED(n) mem>
SLDA	<LONG INT reg> SHL:= <addr>
SLDL	<LONG WHOLE reg> SHL:= <addr>
SRDA	<LONG INT reg> SHR:= <addr>
SRDL	<LONG WHOLE reg> SHR:= <addr>
SP	<DECIMAL(m) mem> -:= <DECIMAL(n) mem>
STC	<CHAR mem> := <CHAR reg>
SXR	<LONG REAL reg> -:= <LONG REAL reg>
UNPK	<ZONED(m) mem> := <DECIMAL(n) mem>
XC	<CHARS(n) mem> XOR:= <CHARS(n) mem>
ZAP	<DECIMAL(m) mem> := <DECIMAL(n) mem>

This accounts for 90 out of 185 instructions, leaving 95:

> AU, AUR, AW, AWR, BALR (non-branching), BC, BCR, BCT, BCTR, BCTR(non-branching), BXH, BXLE, C, CD, CDR, CDS, CE, CER, CH, CL, CLC, CLCL, CLI, CLM, CLR, CLRIO, CP, CR, CS, diagnose, D, DP, DR, ED, EDMK, EX, HDV, HIO, ICM, IPK, ISK, LCTL, LM, LPSW, LRA, M, MC, ME, MER, MR, MVCL, MVN, MVO, MVZ, MXD, MXDR, PTLB, RDD, RRB, SCK, SCKC, SIGP, SIO, SIOF, SPT, SPX, SPM, SPKA, SRP, SSK, SSM, STAP, STCK, STCKC, STCM, STCTL, STIDC, STIDP, STM, STNSM, STOSM, STPT, STPX, SU, SUR, SVC, SW, SWR, TCH, TIO, TM, TR, TRT, TS, WRD

3.15.5 New Predefined Variables

- PSW psw /* the current psw */
- LONG WHOLE clock /* the time of day clock */
- LONG WHOLE clock comparator /* what the clock is compared with */

- LONG INT cpu timer /* an accurate interval timer to measure cpu usage */
- CPUID cpu id
- SHORT INT cpu address
- PREFIX prefix
- fields of psw include: BYTE system mask, PROTECTION_KEY protection key, PROGRAM_MASK program mask, CC condition code, PTR instruction address

On an IBM360, BALR produces the right half OF psw, but on an IBM370, it produces what would be the right half OF psw, if the machine were in basic control mode (mode(psw) = BCPSW).

Having these objects covers the following 15 instructions:

IPK	r2 := protection key OF psw
LPSW	psw := <PSW mem>
SCK	clock := <LONG WHOLE mem>
SCKC	clock comparator := <LONG WHOLE mem>
SPKA	protection key OF psw := <addr>
SPM	program mask OF psw := <PROGRAM_MASK reg>
SPT	cpu timer := <LONG INT mem>
SPX	prefix := <PREFIX mem>
SSM	system mask OF psw := <BYTE mem>
STAP	<SHORT INT mem> := cpu address
STCK	<LONG WHOLE mem> := clock
STCKC	<LONG WHOLE mem> := clock comparator
STIDP	<CPUID mem> := cpu id
STPT	<LONG INT mem> := cpu timer
STPX	<PREFIX mem> := prefix

This accounts for 105 out of 185 instructions, leaving 80:

AU, AUR, AW, AWR, BALR (non-branching), BC, BCR,
BCT, BCTR, BCTR(non-branching), BXH, BXLE, C,
CD, CDR, CDS, CE, CER, CH, CL, CLC, CLCL, CLI,
CLM, CLR, CLRIO, CP, CR, CS, diagnose, D, DP,
DR, ED, EDMK, EX, HDV, HIO, ICM, ISK, LCTL, LM,
LRA, M, MC, MR, MVCL, MVN, MVO, MVZ, MXD, MXDR,
PTLB, RDD, RRB, SIGP, SIO, SIOF, SRP, SSK, STCM,
STCTL, STIDC, STM, STNSM, STOSM, SU, SUR, SVC,
SW, SWR, TCH, TIO, TM, TR, TRT, TS, WRD

3.15.6 New Operators

Although not very attractive, no preferable alternatives have been found to PL360's use of "++" for unnormalized add and "--" for unnormalized subtract. EDIT is basically a move with modification, so EDIT:= is suggested. The remaining multiply instructions are included with the proviso that the mode is INSTRUCTION only if the second operand is the correct (second for integer, first for real) half of the first operand.

ALGOL68C[8] uses :=:= ("is replaced by") for a variety of assignments where the value of (a :=:= b) is the original value of a, rather than its new value. This is also used in an op-and-is-replaced by format; for example, the value of (a := 4; a +:=:= 3) is 4. These operators can express 19 more machine instructions.

AU	<SHORT REAL reg> ++:= <SHORT REAL mem>
AUR	<SHORT REAL reg> ++:= <SHORT REAL reg>
AW	<REAL reg> ++:= <REAL mem>
AWR	<REAL reg> ++:= <REAL reg>
BCTR(non-branching)	<INT reg> -:= 1
ED	<CHARS(n) mem> EDIT:= <DECIMAL(m) mem>
EX	<GENRL reg> EXEC <INSTRUCTION mem>
-	EXEC <INSTRUCTION mem> (==> r0)
M	<LONG INT reg> := <INT reg> * <INT mem>
ME	<REAL reg> := <SHORT REAL reg> * <SHORT REAL mem>
MER	<REAL reg> := <SHORT REAL reg> * <SHORT REAL reg>
MXD	<LONG REAL reg> := <REAL reg> * <REAL mem>
MXDR	<LONG REAL reg> := <REAL reg> * <REAL reg>
MR	<LONG INT reg> := <INT reg> * <INT reg>
STNSM	<BYTE mem> := system mask OF psw AND:=:= <BYTE literal>
STOSM	<BYTE mem> := system mask OF psw OR:=:= <BYTE literal>
SU	<SHORT REAL reg> --:= <SHORT REAL mem>
SUR	<SHORT REAL reg> --:= <SHORT REAL reg>

```
SW       <REAL reg>  --:=  <REAL mem>
SWR      <REAL reg>  --:=  <REAL reg>
```

This accounts for 124 out of 185 instructions, leaving 61:

BALR (non-branching), BC, BCR, BCT, BCTR, BXH, BXLE, C, CD, CDR, CDS, CE, CER, CH, CL, CLC, CLCL, CLI, CLM, CLR, CLRIO, CP, CR, CS, diagnose, D, DP, DR, EDMK, HDV, HIO, ICM, ISK, LCTL, LM, LRA, MC, MVCL, MVN, MVO, MVZ, PTLB, RDD, RRB, SIGP, SIO, SIOF, SRP, SSK, STCM, STCTL, STIDC, STM, SVC, TCH, TIO, TM, TR, TRT, TS, WRD

3.15.7 New Syntax

Let the mode LABEL = REF INSTRUCTION. See section 3.2 for a description of masks. The reasoning behind the choices for BCT, BXLE, and BXH is presented in subsection 8 below. Several operations use only part of a general register. For these cases, we need some way of indicating the desired part of it, a sort of a subscript. The following notation is proposed:

```
<register> .  / <bit> <bit> <bit> <bit> /
e.g.     r1 .  / 1011 /
```

An alternate notation is derived from the ALGOL 68 operator **elem**. In this notation the expression above would appear as, for example:

```
(1,3,4)  ELEM  r1
```

This could act as a source, destination, or comparand.

For referencing several registers in a row, ellipsis notation is convenient. One could also consider that mathematics uses ellipsis where ALGOL uses TO, and allow TO here also. E.g.:

```
(r1,...,r5)
(r14,...,r12)        what we usually save
(r1 TO r5)
(r14 TO r12)
```

Also, one could use one of the array-style notations proposed in the subsection Register Names (3.8.3).

Several operators really have more than two operands. We need a way of having passing 3 or more operands to an instruction. One way is to use the IBMASM notation:

```
<operand> <Operator> <operand>,<operand>,...
```

An improvement would be to allow an operator to have more than one word in it, effectively replacing some of the commas by words. 17 more instructions are presented:

```
BC      GO TO  <LABEL literal>  IF  <mask>
BCR     GO TO  <LABEL reg>  IF  <mask>
BCT     GO TO  <LABEL literal>  IF  COUNT  <INT reg>
BCTR    GO TO  <LABEL reg>  IF  COUNT  <INT reg>
BXH     GO TO  <LABEL literal>  IF  <INT reg>
               XGT  <STRUCT(INT,INT) reg>
BXLE    GO TO  <LABEL literal>  IF  <INT reg>
               XLE  <STRUCT(INT,INT) reg>
D       DIVIDE  <LONG INT reg>  BY  <INT mem>
               PRODUCING  <tag>,<tag>
DP      DIVIDE  <DECIMAL(m) mem>
               BY  <DECIMAL(n) mem>
               PRODUCING  <tag>,<tag>
DR      DIVIDE  <LONG INT reg>  BY  <INT reg>
               PRODUCING  <tag>,<tag>
EDMK    <CHARS(n) mem>  EDIT:=  <DECIMAL(m) mem>  MARK  r1
ICM     <GENRL reg> .  / <4-bits> /  :=
              <CHARS( ) mem>
LCTL    (<control-reg>,...,<control-reg>)  :=
              <ARRAY( )WHOLE mem>
LM      (<general-reg>,...,<general-reg>)  :=
              <ARRAY( )WHOLE mem>
-       regs(m:n)  :=  <ARRAY( )WHOLE mem>
SRP     <DECIMAL(n) mem>  SHIFT:=  <addr>
```

```
                ROUND  <DECIMAL(1) literal>
STCM      <CHARS( ) mem>  :=
                <GENRL reg> .  / <4-bits> /
STCTL     <ARRAY( )WHOLE mem>  :=
                (<control-reg>,...,<control-reg>)
STM       <ARRAY( )WHOLE mem>  :=
                (<general-reg>,...,<general-reg>)
```

This accounts for 141 out of 185 instructions, leaving 44:

BALR (non-branching), C, CD, CDR, CDS, CE, CER,
CH, CL, CLC, CLCL, CLI, CLM, CLR, CLRIO, CP, CR,
CS, diagnose, HDV, HIO, ISK, LRA, MC, MVCL, MVN,
MVO, MVZ, PTLB, RDD, RRB, SIGP, SIO, SIOF, SSK,
STIDC, SVC, TCH, TIO, TM, TR, TRT, TS, WRD

3.15.8 New Procedures

Various forms have been devised for the remaining instructions, but the result is either a procedure or a COBOL-style keyword notation (operators). An implementor could avoid a decision altogether by choosing the keyword notation, and providing the procedures as standard macros. For many instructions, there is little to choose between the two forms. For others, a procedural notation is perhaps the most natural of all the possibilities. A special consideration is for the compare instructions. People are used to seeing infix comparison operators, but these are all boolean, whereas the compare operators here return one of four possible values. For a non-boolean result, a prefix notation is preferred. Four alternatives have been considered for compare operators, but for brevity, they are only presented for "C"; for the rest only one alternative is shown.

Assume the following:

```
MODE STRING = STRUCT( PTR address,
                      PTR WITH (CHAR pad) len );
MODE LINK = PTR WITH (WHOLE(0,2) ilc, cc,
                           PROGRAM_MASK  program
mask);
PROC LINK getlink;
PROC(ADDR)PROTECTION_KEY protection key;
PROC(ADDR)STORAGE_KEY  storage key;
```

Here is a list of the remaining instructions:

```
BALR      <WHOLE reg> := getlink
C         compare ( <INT reg> , <INT mem> )
-         COMPARE  <INT reg> , <INT mem>
-         <INT reg> CF <INT mem>
-         COMP  <INT reg> , <INT mem>
CD        compare ( <REAL reg> , <REAL mem> )
CDR       compare ( <REAL reg> , <REAL reg> )
CDS       test and update ( <LONG WHOLE mem> control word,
                <LONG WHOLE reg> new value,
                <LONG WHOLE reg> old value )
CE        compare ( <SHORT REAL reg> , <SHORT REAL mem> )
CER       compare ( <SHORT REAL reg> , <SHORT REAL reg> )
CH        compare ( <INT reg> , <SHORT INT mem> )
CL        compare ( <WHOLE reg> , <WHOLE mem> )
CLC       compare ( <CHARS(n) mem> , <CHARS(n) mem> )
CLCL      compare ( <STRING reg> , <STRING reg> )
CLCL      compare ( $<STRING reg> , $<STRING reg> )
CLI       compare ( <CHAR mem> , <CHAR literal> )
-         compare ( <CHAR literal> , <CHAR mem> )
CLM       compare ( <GENRL reg>./<4-bits>/ , <CHARS(m) mem> )
CLR       compare ( <WHOLE reg> , <WHOLE reg> )
CLRIO     clear io ( <addr> )
-         CLEAR IO  <addr>
CP        compare ( <DECIMAL(m) mem> , <DECIMAL(n) mem> )
CR        compare ( <INT reg> , <INT reg> )
CS        test and update ( <WHOLE mem> control word,
                <WHOLE reg> new value,
                <WHOLE reg> old value )
diagnose     diagnose ( <bitstring> )
HDV       halt device ( <addr> )
-         HALT DEVICE  <addr>
HIO       halt io ( <addr> )
-         HALT IO  <addr>
ISK       <STORAGE_KEY reg>  :=
                storage key ( <REF AMODE reg> )
LRA       <REF AMODE reg>  :=  real address ( <AMODE reg> )
```

```
-         <REALREF AMODE reg>  :=  @ <AMODE reg>
-         <REALREF AMODE reg>  :=  @@ <AMODE reg>
MC        monitor ( <WHOLE(0,4) literal> , <addr> )
-         MONITOR  CLASS  <WHOLE(0,4) literal>  CODE  <addr>
MVCL      $<STRING reg>  :=  $<STRING reg>
-         move ( <STRING reg> , <STRING reg> )
MVN       move numerics ( <DECIMAL(n) mem>,<DECIMAL(n) mem> )
-         numerics OF <DECIMAL(n) mem> :=
                numerics OF <DECIMAL(n) mem>
MVO       move with offset (<DECIMAL(m) mem>,<CHARS(n) mem> )
-         digits OF <DECIMAL(m) mem> := <CHARS(n) mem>
MVZ       move zones ( <DECIMAL(n) mem> , <DECIMAL(n) mem> )
-         zones OF <DECIMAL(n) mem> :=
                zones OF <DECIMAL(n) mem>
PTLB      purge tlb
-         PURGE tlb
RDD       read direct ( <CHAR mem> , <CHAR literal> )
-         READ  <CHAR mem>  SIGNAL  <CHAR literal>
RRB       reset reference bit ( <addr> )
-         RESET  reference bit ( <addr> )
-         reference bit ( <addr> )  :=  FALSE
SIGP      SIGNAL  <SHORT INT reg> , <addr> STATUS <WHOLE reg>
-         signal ( <SHORT INT reg> , <addr> , <WHOLE reg> )
SSK       storage key ( <REF AMODE reg> )  :=
                STORAGE_KEY reg>
STIDC     get channel id ( <addr> )
SVC       supervisor ( <WHOLE(1) literal> )
-         SVC  <WHOLE(1) literal>
TCH       test channel ( <addr> )
-         TEST CHANNEL  <addr>
TIO       test io ( <addr> )
-         TEST IO  <addr>
TM        test ( <BITS(8) mem> , <BITS(8) literal> )
-         TM <BITS(8) mem> , <BITS(8) literal>
-         test ( <BITS(8) mem> & <BITS(8) literal> )
TR        TRANSLATE  <CHARS(n) mem>  BY  <CHARS(256) mem>
-         TRANSLATE  <CHARS(n) mem>  WITH  <CHARS(256) mem>
-         TRANSLATE  <CHARS(n) mem>  USING  <CHARS(256) mem>
-         TRANSLATE  <CHARS(n) mem>  VIA  <CHARS(256) mem>
-         translate ( <CHARS(n) mem> , <CHARS(256) mem> )
TRT       SCAN  <CHARS(n) mem>  USING  <CHARS(256) mem>
                MARK  r1, r2
-         SCAN  <CHARS(n) mem>  WITH  <CHARS(256) mem>
                MARK  r1, r2
-         SCAN  <CHARS(n) mem>  VIA  <CHARS(256) mem>
                MARK  r1, r2
-         SCAN  <CHARS(n) mem>  BY  <CHARS(256) mem>
                MARK  r1, r2
-         trt ( <CHARS(n) mem> , <CHARS(256) mem> )
TS        test and set ( <BYTE mem> )
WRD       write direct ( <CHAR mem> , <BYTE literal> )
-         WRITE  <CHAR mem>  SIGNAL  <BYTE literal>
```

3.15.9 Support for BCT, BXLE, BXH

There are two basic approaches which one could take to these instructions. One could enumerate which special cases of a general form are implementable using these more efficient instructions; then the programmer who cared would use these forms and the compiler would look hard for them. Or, one could design special forms so that both programmer and compiler could easily recognize the intended use of these instructions.

Example:

```
general form:
   IF <boolean clause> THEN <clause> FI
special case:
   <boolean clause> ::= <var in genrl reg> -:= 1
   <clause> --> GO TO <label>
special code:
       BCT <genrl reg>,<label>
```

Example:

```
general form:
   [FOR <tag>] [FROM <int-unit1>]
     [BY <int-unit2>] [TO <int unit3>]
     DO <clause> OD
special form:
   -  compiler has a working register available
      OR <tag> is a general register
   - <int unit2> is omitted or is -1
   - <int unit3> is 1
special code:
      <reg> := <int unit1>;
   1: <clause>;
      BCT(<reg>,1)

special form:
   compiler has an available general register
     pair + another general register   OR
      ((FROM part gives a register) OR (FOR part
     gives a register) AND compiler has a
     general register pair)   OR
      ((BY part gives an even general register)
     AND (TO part gives the matching odd
     register) AND (compiler has another general
```

```
            register))   OR
            etc.
       special code:
            <int unit1>;
            <int unit2>;
            <int unit3>-1;
         t: BXH(<reg>,f);
            <clause>;
            GO TO t;
            f: ---
```

A compromise example is the use of a special procedure, such as a PROC(GENRLREG) BOOL decr to zero.

Proposal

A special form to support these three operations via conditions for IF statements was developed. This means that the programmer must build his own loops with initialization parts and jumps.

(a) <boolean primary> ::= COUNT <general register usage>
This operator decrements the register and returns true if it is non-zero.

(b) <boolean primary> ::= <general register usage> XGT <general register usage>
This operator adds the second general register to the first general register. It then returns true if the value of the first general register is now greater than the value of the second general register (the value of the second general register+1 if the second general register is even).

(c) <boolean primary> ::= <general register usage> XLE <general register usage>
This operator adds the second general register to the first

general register. It then returns true if the value of the first general register is now less than or equal to the value of the second general register (the value of the second general register+1 if the second general register is even).

These three boolean primaries can be used anywhere a relation could be used; e.g.

```
IF  COUNT r1 AND x < 7  THEN ...
r4 := n+1;
WHILE  COUNT r1  DO ...  OD;
```

Since, unlike a relation, the negation of these primaries cannot be tested, some uses (e.g. the first above) may involve conditionally skipping a jump and may be no more efficient than a more straightforward usage.

3.16 Declarations

In Algol 68 one must read to the end of a range in order to begin to parse it; this necessitates another pass which increases compilation costs. This cost income can be eliminated by requiring everything to be declared before use. Mutual recursion at the same level (e.g. procs or modes calling each other) and forward branches before any references can be permitted by a half-declaration before any references to the other are made. For example:

```
ALLOW PROC(INT) REAL p ,  LABEL l ;
PROC q = (INT x) REAL :
   IF  x > 0  THEN  7.0  ELSE  p(x+1)  FI;
PROC p = (INT x) REAL :
   IF  x > 0  THEN  6.0  ELSE  q(x+2)  FI;
GOTO l ;
...
:l: SKIP
```

Many ALGOL 68 implementors have argued against reserving all mode names in any reserved word scheme for that language because of the asymmetric nature of a workable scheme. A tag X in an outer block is shielded by a mode X in an inner block, but not vice versa. This asymmetry is removed, and parsing simplified, if all mode indicants and operator indicants are fixed before object declarations begin.

There are two major styles for declarations. In ALGOL and FORTRAN, the type or mode is mentioned, and then a list of identifiers may follow. In IBMASM, COBOL, and PL/I (without factoring), each identifier is presented separately and then its attributes are given. This second form is more verbose, but the identifiers are more visible. It also allows several attributes to appear without hindering readability, whereas the first scheme suffers if the type name is more than one word. I prefer PL/I's style of declarations, but did not use them because changing the style of declaration would be a very large break from ALGOL 68 compared to the benefits.

Pascal introduced subranges of a type as a new type.

The reasoning behind this is that an integer variable may possess any integer value, but many integer values are supposed to possess values only in a limited subset of the possible values. Reliability is enhanced if the programmer can be sure that his variables do possess values only within their intended ranges. For example, a variable 'hourly salary' might be restricted to the range '(minimum wage : 2500)'. Considering Habermann's criticism[16] and Lecarme and Desjardin's response[22], it is clear that the subrange is not really a new type, but rather a request for an automatic range check whenever necessary. This check can be requested in terms of a mode suffix. It can be allowed both in a mode definition and in a variable declaration. E.g.:

```
MODE  DIGIT  IS  INT RANGE(0:9)
INT RANGE(2,3,5,7,11,13,17,23,29)  small prime
```

'Small prime' will be a very expensive object to assign to. The range suffix is given with the word "RANGE" rather that being simply appended to the mode (e.g. INT(0:9)) because parentheses after mode indicants are already used for length modifiers.

It would be useful to have some way of describing objects stored at fixed offsets in the machine, for example, the timer at location 80. This can be done by use of register 0 as a base register. E.g.:

```
RECORD  lowmem (
        ¢ ...  stuff for first 80 bytes ...  ¢ ,
        INT  timer,
        ¢ ...  more declarations ¢
              );
BASE  lowmem ON r0;
```

MARY and SUE allow location modifiers on the declaration. The MARY declaration has the form:

```
INT timer AT 80;
```

For modes where packing is a concern, this must be indicated in the mode. Programmer-defined types will usually have less then 257 different values and will fit in a single byte. But on some machines, objects in fullwords can be manipulated faster than objects in single bytes. There must be some default, either packed or expanded. I believe the best default is expanded just enough for fast loading. BOOLs will normally be stored one per byte, because byte addressing is easier than bit addressing. Programmer-defined types can be stored in a halfword, the smallest unit which can be loaded into a register in one instruction. Registers are a consideration because one may increment or decrement a variable of a programmer-defined type. These variables can also be stored in packed decimal format, using one byte if there are 10 values or less.

A programmer may wish to pack an object more or less tightly than the default. He should be able to indicate the size of the area of memory in which an object will be stored, and where in that area it will be stored. E.g.:

```
BOOL  IN  WORD        ¢ in fullword
BOOL  RJUST  IN  BYTE    ¢ last bit in a byte
BOOL  LJUST  IN  HWORD   ¢ in halfword
MODE C = colour  IN  HWORD
MODE YR = YEAR  EXCESS -1900
```

A full procedure declaration repeats much information.

This is because the full mode appears on the left and the mode of each parameter appears on the right. ALGOL 68 makes a special case to shorten procedure declarations. One may abbreviate:

```
PROC (<arg mode.1>,...,<arg mode.n>)<resmode>
   <tag>  =  (<arg mode.1> <arg id.1>, ...,
               <arg idn> ) <resmode> :
      <routine body>
CORP
```

to:

```
PROC  <tag>  =
   (<arg mode.1> <arg id.1>, ...,
    <arg idn> ) <resmode> :
      <routine body>
CORP
```

This assumes that the mode of the actual procedure value is correct. Another abbreviation is borrowed from MARY.

```
CONS <mode> <tag>  =  <value>
```

may be abbreviated to

```
<mode> <tag>  =  <value>
```

which is distinguished from

```
<mode> <tag>  :=  <value>
```

which makes a variable. This abbreviation that a missing or extra ":" may be undetected, but the perturbed source program is very close to the intended one. This serves to reduce the clutter in a program.

3.17 Expressions

Having determined what the basic actions are, and how to combine units into clauses, it remains to fill in the definition of a unit. Although variations can be made there are two fundamentally different possibilites:

(a) Any ALGOL 68 expression (within reason) will be accepted. This implies an expression has similar semantics to the same expression in ALGOL 68, that expressions are evaluated in the same order (sometimes left-to-right and sometimes right-to-left), and that there is an operator priority mechanism. This means that the compiler is prepared to generate many temporary variables.

(b) Only the same kinds of expressions that PL360 accepts will be accepted. This means:

- the left hand side of an expression is a very simple expression.
- No expression uses a register not mentioned. This means that unless the machine can evaluate the expression in memory, the destination is a register in which it can be evaluated.
- no brackets - strict left-to-right evaluation of operators

This choice is very important. It determines whether PL370 is a high-level language or a low-level language. I have chosen (a) because I believe it will make PL370 a more useful language and a more convenient language. It also makes it a more expensive language to compile, because the compiler will be doing the register management work that the programmer does in PL360 and Assembler.

The proviso "within reason" must be elaborated. I want to keep out of the language those features which would prohibit one-pass compilation. Therefore, while a source-expression may be complex, a destination-expression is severely restricted. It may not be conditional or involve any operators except that one may indicate an address expression by the monadic dereferencing operator and follow that by a complex expression. Otherwise, a destination is a variable, possibly including selectors or subscripts. Without this restriction is would be difficult for the compiler to determine when to produce values and when to produce addresses (without seeing the ":=").

A second proviso is that if a PL370 assignment statement is simple enough to be acceptable under (b), then the code produced must be as good as it would have been if (b) had been chosen. Example:

source	basic actions

r1 := r2 + r1	tempreg := r2;
	tempreg +:= r1;
	r1 := tempreg

r1 := r2 + r3	r1 := r2; r1 +:= r3;

In ALGOL 68, a function call may appear in an expression. This may happen in PL370 also, provided that "function" is properly defined. Languages with a single linkage convention usually include functions, but those without, don't. The problem is to locate the value of a function. All expressions in programming languages are single-valued. This means that if a procedure returns more than one value and it is used as a function, one value must be used and one ignored, or the two combined by some rule into a struct. In order to avoid confusion, and because adding functions is only to make programs more concise, not to increase power, only single-valued procedures may be functions.

definition : A user-written function is a procedure which has exactly one result. This may be either an OUTPUT register or a RESULT parameter in memory.

This definition allows the user to write functions using any linkage convention. Unfortunately, this works in a non-intuitive manner for S-type linkage: those procedures which return a result may not be functions, while PROC (...)VOID's can. This is because these procedures always return a return code in addition to any value.

3.18 Compile-time Actions

3.18.1 Varieties of Compile-time Actions

In many cases, when a compromise is needed between high-level and low-level languages, the technique used to find a mid-level alternative is to parameterize some higher-level construct (e.g. ARRAYTYPE). But other constructs (e.g. ENTRY sequence) cannot be built that way without undue loss of freedom - the user must be given complete control over that phase of the semantics. Thus, PL370 without some sort of macros is unthinkable. But what preprocessing should be available? Consider the following:

- straight text substitution without parameters (like XPL,C)
- straight text substitution with parameters (proposed for PL360)
- variables may exist at compile-time. Text macros may also play with these variables (like IBMASM)
- syntax macros. As with a text macro, it produces some text, but the resulting text must be source of a specified mode.
- as above, but with the ability to type check operands
- compile-time procedures which produce some compile-time action
- as above, but they may produce some source text as a side effect
- as above, but the text is of some fixed mode.

Certainly compile-time objects should exist. When eventually used they can turn into denotations or pieces of text. The usual macro-processor attached to a compiler has a "compile-time language" which is higher level than the "ordinary" language. It seemed reasonable to follow the PL/I model and make the compile time language very similar to the run-time language, with most of the extensions to ALGOL 68 left out. As in PL/I, a compile-time action can be indicated as a special character prefixing the statement. This must be an otherwise useless character. "¦" was considered as it is easy to input and would never be missed (the PN print chain excludes "¦") if the compiler listed the program after pre-procesing. However, a printing character is safer and PL/I's "%" is as good as any.

There are 6 types of "macros" which might be provided:

(1) %PROC - accepts compile-time values and produces a compile-time result. This is called like a proc (as in PL/I).

(2) MACRO[1] - accepts source text as arguments and has no explicit result, but the body is expanded to source code (as in IBMASM). The macro body will be source text and will be delimited by the end of the macro header and a terminating symbol, such as MEND. This is called like a proc. An alternate form of definition is that used by FMT [25], in which the macro has no formal parameter list and the body may ask for any actual parameter by position number.

(3) %OP - like %PROC, and operates on compile-time values. It is called like an ordinary operator, but there is no longer any in-line/out-of-line distinction. %OP could be

used as nice frill, but as few mode definitions are expected in the compiler-time program, it would be of marginal use. Also, an operator could become truly overloaded.

Example. Suppose "+" is defined on compile-time integers. Consider the source text:

```
r := m * 3 + 4
```

A likely interpretation would be:

```
L  r,m
M  r,3+4        i.e.  M    r,7
```

So, %OP is more trouble than it is worth.

(4) MACRO² - accepts source which represents a requested run-time mode. This can be simulated by MACRO¹. For example:

```
MACRO² x ( INT a, REAL b ) BOOL = ...
   ( ...  a ...  b ...  )
          or
MACRO¹ x ( a, b )   =
   BOOL ( ...  INT ( a ) ...  REAL ( b) ....  )
```

This will make the parameters of the required mode (possibly by run-time action) or will lead to an error during ordinary processing. It is also much uglier if the macro is ever expanded.

(5) MACRO³ - similar to MACRO¹, but instead of procedural style call, the macro header gives a pattern to match for.

(6) MACRO⁴ - similar to MACRO², but called like MACRO³.

Actually, MACRO¹ can be made a special case of MACRO³, and MACRO², a special case of MACRO⁴. MACRO⁴ would be nice, but appears to have hidden traps, e.g. after:

```
REAL  x; ( mac(x,y)
```

Suppose that 'mac' has generated, so far:

INT x

and then it calls 'mac2'. What is the mode of 'x'? If the next symbol generated is a delimiter ("," or ";"), then 'x' is being redeclared, but if it is a letter, then this declaration does not affect 'x'.

{ Problem. What kind of operand can a macro take? If it is an arbitrary symbol, then things are not too bad, although the symbol could be moderately long. But if it is arbitrary text, then that can be very long. This problem occurs with all forms. A possible solution could be that as soon as it is detected that this is more than a single symbol, the whole actual parameter is swallowed up as the definition of an implicitly defined macro, and the actual parameter of the original macro is a call on the new macro. }

Conclusion.

%PROC should be included. MACRO^3 can be included as long as the pattern matching is fairly simple (e.g., match for keyword and swallow parameter until next keyword).

3.18.2 How To Distinguish between Compile-time Actions and Run-time Actions

In IBMASM, assembly-time statements are distinguished by a different set of operators. In PL/I, conditional actions are triggered by a "%". Inside a %PROC, all actions are compile-time and source code is put into %CHARACTER variables. But outside a %PROC, what happens? For example, suppose a programmer wants:

```
IF target machine is a 370
 THEN   some source code
 ELSE   other source code
FI
```

This must be distinguished from:

```
IF target machine is a 370
 THEN   some compile-time statement
 ELSE   other compile-time statement
FI
```

There are two possible regimes:

(1) Assume run-time whenever possible. Thus, %IF, %THEN, %ELSE, and %FI would all be marked as would the actions in the second example. The boolean clause after a %IF, obviously does not need marking - it must be evaluated at compile-time.

(2) As soon as a compile-time clause is entered (e,g, %IF, %PROC, %CASE), the whole clause is done at compile-time. One possible compile-time statement is a source-code generator which causes some text to become source text (after pre-processing). Several forms were considered, of which the best two are:

(a) << text >>

The text is scanned and any compile-time variables are replaced by their values. The text then is processed as source.

(b) A procedure called generate (or gen for short) which behaves just like ALGOL 68's print, except that it writes on the "source file". This form is more precise than the other because the programmer presents string denotations which are not modified, and symbols which must be compile-time expressions. The programmer may also use all the ALGOL 68 layout features (newline, newpage, set char number, etc.) to beautify his generated source text. This form also allows the pre-processor to be used as a general-purpose macro processor.

For example:

```
        gen ( "INT r = ", m * q, ";", newline,
              "REAL mpg; ", newline );
```

It was decided to use form (a) since it is more convenient, but to allow form (b) also, as it is more precise, the layout features may be useful, and it introduces no new syntax.

3.19 <u>Separate Compilation Mechanisms</u>

The larger a system, the more crucial separate compilation becomes. Therefore, a systems programming language must facilitate separate compilation. However, PL370 is trying to clean up the interfaces in order to reduce errors, and compilation/assembly boundaries are dirtiest interfaces around. There is no point insisting on all kinds of fancy mode checking if it can all be circumvented by separate compilation. The only safe compromise is to have a new linkage editor which performs all the same mode checks when it matches external symbols, and only produces a loadable program when all references (except to system subroutines) have been resolved.

There are two ways to build systems: bottom-up and top-down. When building bottom-up, a procedure must assume no more environment than it has a right to - at most the standard operating system. It then declares all the external objects it needs (except those assumed in the system). Such a procedure may in turn be declared external and called by any other procedure.

When building top-down, a procedure marks a spot "to be defined later". The compiler then saves the environment as known at that spot and compiles a hook to call the future module. A subsequent module is then compiled as if it were originally in that spot - none of the old variables it uses need be redeclared. Naturally, this module may be called only by the one which left a hole for it in the first place. This method is taken from that used by ALGOL68C [8].

Since the a program communicates with the operating system by procedures (via BALR or SVC), and since all procedure calls are checked for mode mismatch, all the operating system procedures used, must be declared. Since this can be a lot of work, the standard environment for an installation should have all these procedures pre-declared. This is both easier for the programmer, and safer, as he cannot declare one incorrectly.

3.20 Capabilities Restriction

Not all features of PL370 will be equally desirable; some will be ugly, unsafe, but necessary. MARY has made a sharp break between SAFE MARY and UNSAFE MARY, but it seems that anyone can use UNSAFE features.

I suggest that the normal mode of sticking to a safe subset should be enforced by the compiler. Unsafe features should be considered errors unless the programmer has labelled the whole module UNSAFE. Even further, in a business organization, a manager could create profiles for his programmers indicating whether or not they are allowed to specify UNSAFE.

A different method of protection is to banish UNSAFE features to macros written by the better programmers; for a macro, the capabilities could be attached to the library rather than using those of the invoking module.

An obvious candidate for exclusion from the safe subset is GO TO and statement labels. Another possible exclusion is the LINKAGE definition.

3.21 Support For Languages Other Than English

Since readability and understanding are prime goals of this project, it would be counter-productive to insist on the use of English by a person who does not speak and write it well. Since it is a Canadian product, PL370 should be bilingual - English and French. This means that a processor should be able to communicate with the programmer in French as well as it does in English. This could be done by having one bilingual processor, or two equivalent unilingual processors. In the latter case, use of the other language would be illegal. In the former case, the list of keywords would double in size to allow either French or English at any point. The language the processor used for its output could depend in some way on the input, or be strictly one or the other according to an environment declaration. The advantages of this method are that one can never choose the wrong processor; a bilingual programmer may find it advantageous to have two sets of brackets (e.g. IF -- THEN

-- SI -- ALORS -- FSI -- FI may show program structure more clearly than IF -- THEN -- IF -- THEN -- FI -- FI); and program fragments can be more easily combined. The disadvantage is the same as the second advantage; those easily written bilingual programs may be hard to read for unilingual readers, who will be forced to learn twice as many keywords in order to be able to read a program. Actually, the acceptance of French keywords is a minor problem; one can make a library of synonym keywords for any other language. The real problem is to get the processor to talk back in French; this cannot be patched from the outside unless the processor has a file containing all possible messages, none of which is variable, which can be replaced by an installation.

3.22 Use of Extra Graphic Symbols

Extra symbols make the reading of a program easier if the symbols are familiar. Although it is right not to require any more characters than a PN print train will provide, most languages also do not allow any such characters. However the TN and APL print trains, when available, provide many useful symbols. Both the TN and the APL print trains provide "≠", "≥", "≤", "[", "]", "{", and "}". The TN train provides "±" and superscripts. The APL train provides a multiplication-symbol, a divide-symbol, floor and ceiling symbols, and four arrows. Thus it would be desirable to allow as many symbols as possible. It would also be desirable to ensure that only printable symbols are used; however, a compiler cannot tell where the listing will be printed if it is sent to a file or spool pack. In fact, printability can be checked[1] only if the listing is made directly onto a printer, if the system will tell a user what train is mounted, and if the compiler knows about that kind of print train (including all local idiosyncracies). This is obviously very unlikely. Thus, when extra characters are allowed by the compiler, it must trust the programmer to ensure that all are visible. In my opinion, the advantages of better readability for the careful outweigh the disadvantages of lost readability for the careless. There is another disadvantage of using these characters: they limit the portability of a program to those installations which have an acceptable printer.

The usual technique of "teaching" compilers is a declaration. A possible example is:

```
    NEW OP: ≠, ≥, ≤, +, ¯, ×.
```

Operators are easy to add like this, provided that they may not be combined into multi-symbol operators (like "<="). They may then be used freely both dyadically and monadically. Other symbols may seem unnatural as operators, e.g. "{". Since brackets are usually in short supply, it would be useful to allow the user to add pairs or brackets. These would be equivalent to "(" and ")". It is also possible to add letters (e.g. the hyphen "-"). Possible declarations:

---

[1] This is not true for control characters, whose codes are less than the code for a space. Control characters are never printable, and may be forbidden as symbols in the language.

```
     NEW BRACKETS: ⁽, ⁾; {, }.
     NEW LETTERS: -.
```

Digits are harder because each one has some semantic meaning. Even knowing that "^{8}" is a digit symbol is not enough; should 34^8 = 348? Allowing x^2 for x**2 improves readability (in those rare cases when one needs exponentiation), but any more support for superscripts would be out-of-place in a lower-level language such as PL370.

CHAPTER 4

FINAL DESIGN OF THE LANGUAGE

In this chapter, the form " { x }* " means 0 or more repetitions of whatever x is. The form " { x | b | ... | d } " means that a choice of one of x or b or ... or d is mandatory. The form " [x] " means that x is optional. The form " [x | b | ... | d] " means that a choice of one of x or b or ... or d is optional.

4.1 Data

4.1.1 Types and Modes

Each datum has a type which determines the set of values it may possess. The type of a datum is an idealized set.

There are 7 basic types:

integer - the set of integers

real - the set of floating-point numbers
(defined by pairs of integers)

bits - uninterpreted sequences of bits

bool - { TRUE , FALSE }

character - single characters

whole - the non-negative integers

instruction - single machine instructions

Types are broken down into modes, which are more specific as to representation of information and datum size. The set of values which may be represented by an object of a mode is a subset of the type of that object. The type of a datum determines the set of operators which may access it. The basic modes are derived from the basic types as follows:

from integer :

INT	signed 4-byte binary integer (2's complement)
SHORT INT	signed 2-byte binary integer (2's complement)
LONG INT	signed 8-byte binary integer (2's complement)
INT(n)	signed binary integer, n bytes long
INT(0,m)	signed binary integer, m bits long, $1 \le m \le 7$
DECIMAL(n)	a signed, packed decimal number, n bytes long, $1 \le n \le 16$ i.e. 2*n-1 digits long
ZONED(n)	a signed, unpacked decimal number, n bytes long, $1 \le n \le 16$ i.e. n digits long

from _real_ :

REAL	signed 8-byte hexadecimal floating point
SHORT REAL	signed 4-byte hexadecimal floating point
LONG REAL	signed 16-byte hexadecimal floating point

from _bits_ :

UNKNOWN(n,m)	an uninterpreted object, n bytes + m bits long

from _bool_ :

BOOLEAN(t,f)	one character (t for TRUE,f for FALSE)
BOOL	BOOLEAN("T","F")
BIT	one bit, 1 or 0

from _character_ :

CHAR	one character (one byte), usually EBCDIC

from _whole_ :

WHOLE	unsigned 4-byte binary integer (i.e. ≥ 0)
SHORT WHOLE	unsigned 2-byte binary integer
BYTE	unsigned 1-byte binary integer
NIBBLE	unsigned 1/2-byte binary integer
WHOLE(n)	unsigned binary integer, n bytes long
WHOLE(0,m)	unsigned binary integer, m bits long, $1 \leq m \leq 7$

DISPLAY(n)	an unsigned, unpacked decimal number, n digits long, 1 ≤ n ≤ 16
DISPLAYB(n)	same as DISPLAY(n), but may contain leading blanks and may have a sign character

from *instruction* :

INSTRUCTION(n)	an instruction of 'n' halfwords, 1 ≤ n ≤ 3

In addition, one may define new types, and derive further types from old ones. The derived types include pointers, arrays, structures, records, and unions.

Programmer-defined Types These are borrowed from Pascal. One mentions a list of new identifiers, which comprise all the valid values of that type. For example,

TYPE colour = (blue,green,red,yellow,orange)

defines a type with exactly 5 values. This implicitly defines a mode called COLOUR. Other modes may be defined from a type if the size of the default mode is not suitable.

definition : finite type A type is a finite type if it is integer, whole, bool, or character, or if it is a programmer-defined type or a set type.

Sets Sets are also borrowed from Pascal (where they are called powersets). For any finite type "ffff", "set of ffff" is a type. The possible values for a "set of ffff" object are the different subsets of the set of all possible "ffff" values. A set type is implemented by a bitstring of as many bits as there are different values of the base type. The basic operations on sets are: adding members, deleting members, testing for membership, uniting, and intersecting.

Pointers Pointers are machine addresses. The user must specify what mode of object is being pointed at.

There are three basic address modes. A REF pointer will have its first byte zero on machines with 24-bit addressing (this allows easy comparison). A PTR pointer is the last three bytes of a word, allowing any value in the first byte. A SHORT REF pointer is a halfword.

Arrays An array is a contiguous sequence of objects all of the same mode. Multi-dimensional arrays are allowed. The bounds of an array are determined at compile-time. The bounds for each dimension are an upper and lower bound of the same finite mode; integer bounds may also include a BY part. One may subscript by giving as many subscripts as the array has dimensions, e.g. " x(2,4) ". Arrays are classed into "array types" on the basis of storage order, subscript checking, and default lower bounds. For those who want the safer ALGOL construct of an array with a descriptor, a ROW is the descriptor for an array, and may be used just like an ALGOL 68 row.

Records A record is an object which contains one or more sub-objects. It is used solely to indicate hierarchy. It corresponds roughly to a CSECT of data.

Structures A structure is similar to a record in declaration, but is merely a map or template for describing an object. There may be many objects describable by the same structure mode at any one time.

Unions A union is a set of one or more modes. It is used to indicate that the value of an object may be of any one of the modes in the union. No manipulations on the object (other than copying) are allowed without knowing which of the possible modes it is. There are safe unions, for which the compiler generates code to manipulate union flags, and unsafe unions, for which the user provides routines for selection and uniting.

Strings There are at least 11 different ways of representing a string variable on the IBM370, made up of variations on the existence, length, and position of length and address desciptors for the characters. Therefore, the user may redefine STRING so as to fit his needs. The standard definition is:

```
STRUCT ( PTR address, PTR WITH (CHAR pad)
length)
```

4.2 Declarations of Declarers

4.2.1 Types

New types are defined by presenting a list of identifiers which are the values of that type. These identifiers may not be used for any other purpose in the source module. E.g.:

```
TYPE egg size =
        (small,  medium,  large,  extra  large,
jumbo)
```

4.2.2 Linkage Conventions

Each procedure must be declared to be of some LINKAGE convention. Each LINKAGE convention is defined by specifying how the procedure uses the registers of the machine. The LINKAGE convention specifies only where the argument list, if any, can be found in memory, not its format; this format is specified in each procedure declaration. There are six parts to a LINKAGE convention. INPUT specifies what registers are required to contain useful information when the procedure is entered. All registers will be restored by the procedure except those mentioned by NORESTORE or OUTPUT; these two lists must be disjoint. The latter list specifies that the register will contain a value of interest to the caller; the former specifies that the value is no longer meaningful. A register in the OUTPUT list is set by the final unit of the procedure, which is a list of as many values as are needed, and in the same order as in the OUTPUT list. A register in the NORESTORE list need not be changed at all; but if changed, will not be restored. SAVEAREA defines a mode to be used as a savearea by procedures of this LINKAGE convention. CALL specifies a macro to use for calling procedures of this LINKAGE convention. ENTER specifies a macro to use to expand the definition of procedure entry. EXIT specifies a macro to use to expand the definition of procedure exit. These macros may reference many built-in compile-time procedures; for example, one may examine the modes of parameters of the procedures. See Section 4.7 for an explanation of the macro language.

```
<linkage-declaration> ::= LINKAGE  <linkage-indicant>  =
           INPUT ( <reg-syn>  { , <reg-syn> }* ),
           NORESTORE ( <register> { , <register> }* ),
           OUTPUT ( <reg-syn>  { , <reg-syn> }* ),
           SAVEAREA = <mode-indicant>,
           CALL = <macro>,
           ENTER = <macro>,
           EXIT = <macro>
<reg-syn> ::= <mode-indicant> <register>
       | <mode-indicant> <tag> = <register>
```

In this context one may also use the 'modes' ENTRY, RETURN, and PLIST on the INPUT list. ENTRY means the entry address of the procedure. RETURN means the return address. PLIST means the parameter list in memory; this requires the parameters or their addresses to be contiguous.

Built-in linkages include S_TYPE, R_TYPE, and LOCAL. A sample linkage declaration is shown in Appendix II.

4.2.3 Arraytypes

Arraytypes are declared as described in Chapter 3.

Example:

```
ARRAYTYPE ARRAY = (ROW-MAJOR,CHECK,1);
ARRAYTYPE DIMENSION = (COLUMN-MAJOR,NOCHECK,1);
```

4.2.4 Modes

4.2.4.1 Basic Declarers

By default, for each programmer-defined type, a mode will be created with the same name except that the letters are in upper case and blanks are deleted. This mode uses one fullword for each datum.

Modes can be defined in terms of other modes. This form is used to define structures, array modes, reference modes, union modes, procedure modes, etc.

```
<mode-declarer> ::= <basic-mode-declarer> /* from 4.1.1 */
      | <type-name>
      | SET OF <mode-indicant>
      | REF <actual-mode-declarer>
      | PTR <actual-mode-declarer>
      | SHORT REF <actual-mode-declarer>
      | <array-declarer>
      | ROW ( <dimensions> ) <actual-mode-declarer>
      | STRUCT ( <fields-list> )
      | <united-mode-declarer>
      | <unsafe-united-mode-declarer>
      | <mode-declarer> <mode-modifier>
<fields-list> ::= <fields>  { , <fields> }*
<fields> ::= <actual-mode-declarer> <identifier-list>
<actual-mode-declarer-list> ::= <actual-mode-declarer>
         , <actual-mode-declarer>  { ,
         <actual-mode-declarer> }*
<array-declarer> ::= <arraytype-indicant> ( <dimensions> )
         <actual-mode-declarer>
      | CHARS ( <dimensions> )
      | BITS ( <dimensions> )
<dimensions> ::= <dimension>  { , <dimension> }*
<dimension> ::= <integer>
      | <denotation> : <denotation>
<united-mode-declarer> ::= UNION (
         <actual-mode-declarer-list> )
      | ROW ( <formal-bounds> ) <actual-mode-declarer>
<unsafe-united-mode-declarer> ::= <array-declarer>
      | UNSAFE UNION ( <actual-mode-declarer-list> )
         [ SELECT( <identifier> ) = <macro> ]
         [ UNITE( <identifier> ) = <macro> ]
      | CHARS
```

Examples:

```
     REF CONS INT
     ARRAY(1:5) STRUCT( INT x )
     ARRAY (small:large, 1:20, 2 TO 34 BY 4) EGG
     SHORT REF ARRAY(3) ROW(5) BOOL
     SET OF BOOL
     SET OF SET OF COLOUR
     UNSAFE UNION (CHARS(8),INT,REF FDUB)
        SELECT(x) =
           IF x.(0|3) = 16r000000
            THEN INT
           ELIF x.(0|1) = 16r00
            THEN REF FDUB
            ELSE CHARS(8)
           FI
       UNITE(u) = u
```

4.2.4.2 Mode Modifiers

An <alignment> is a phrase which ensures that the address of the following datum is aligned as desired. The <alignment>s are:

@DOUBLE (multiple of 8) or @DOUBLE+n (0<n<8)

@WORD (multiple of 4) or @WORD+n (0<n<4)

@HALF (multiple of 2) or @HALF+1

@BYTE (on any byte boundary)

@NIBBLE (on byte or half-byte boundary)

@BIT - unaligned

```
<mode-modifier> ::= RANGE ( <values> { , <values> }* )
      | [ LJUST | RJUST ]  IN <word-size>
      | <alignment>
<values> ::= <denotation>
      | <denotation> { : | TO } <denotation>
<alignment> ::= @ <word-size>  [ + <whole-number> ]
<word-size> ::= { DWORD | DOUBLE }
      | WORD
      | { HWORD | HALF }
      | BYTE
      | NIBBLE
      | BIT
<actual-mode-declarer> ::= CONS <mode-declarer>
      | [ VAR ]  <mode-declarer>
```

4.2.4.3 Mode Declarations

The actual mode declaration comes in two kinds. In the normal case, the mode being defined is a new mode, which inherits all the operators and procedures defined on the right hand side mode in the surrounding environment, but shares nothing else. Denotations for this new mode must appear in a cast. One may also use "IS" to request that the new mode be made equivalent to the old mode.

```
<mode-declaration> ::= MODE <mode-indicant> <new sym>
          <mode-declarer>
<new-sym> ::= IS
      | =
```

Modes can be partially parameterized. That is, where a mode indicant is required in a declarer, one may use a mode-indicant parameter, and where an integer would be required in a declarer, one may use an ordinary parameter.

```
<mode-declaration> ::= MODE <mode-indicant> (
          <mode-declaration-parameter-list> ) <new sym>
          <mode declarer (involving parameters)>
<mode-declaration-parameter-list> ::=
          <mode-declaration-parameter>
      | <mode-declaration-parameter-list> ,
          <mode-declaration-parameter>
<mode-declaration-parameter> ::= <bound-parameter>
      | <mode-parameter>
<bound-parameter> ::= % <identifier>
<mode-parameter> ::= <indicant>
```

Examples:

```
MODE EGGSIZE = egg size  ¢  automatic
MODE  A  =  egg  size         ¢     length is one
word
MODE B = ARRAY(n) A        ¢ a vector  of  length
"n"
MODE C = STRUCT ( B fsel1, fsel2, A aa);

MODE D(%n) = ARRAY (%n) INT;
  use of D:
D(7) dd          means [7]INT dd
MODE E(M,%n) = ARRAY (%n) M;
  use of E:
E(INT,7) ee     means ARRAY(7) INT

MODE FBOOL = BIT RJUST IN WORD
MODE DIGIT IS WHOLE RANGE (0:9)
MODE PACKDEC IS DECIMAL(8) @DOUBLE
MODE PPP = UNKNOWN(3,0) RJUST IN WORD @WORD+1
```

4.2.5 Forward Declarations

Because everything must be declared before use, there must be a mechanism to allow the declaration of mutually recursive modes.

```
<forward-declaration> ::= <forward-mode-declaration>
      | <forward-object-declaration>
<forward-mode-declaration> ::= MODE <indicant>
      | OP <indicant>
```

4.2.6 Denotations

4.2.6.1 Plain Denotations

The denotations for programmer-defined types are presented in the declaration. Denotations for strings, reals, bits, and bools are as in ALGOL 68. Denotations for integers are the same as signed ALGOL 68 ints. Denotations for wholes are the same as unsigned ALGOL 68 ints. Denotations for chars are '<char>'. In the case where a type contains more than one mode, the mode of the denotation depends on the context. If the denotation is the only element in an expression, then its mode is that required by the left hand side; otherwise, the mode of the denotation is the main mode of the type.

Examples:

```
LONG REAL x;
REAL y;
```

```
x +:= 1.0; ¢ here 1.0 is LONG REAL
y +:= 1.0; ¢ here 1.0 is REAL

/* string */   "hello"
               "this string contains one ""."
/* bits */   16r34234abd
/* bool */   TRUE , FALSE
/* whole */   15, 100000035
/* integer */   +1, +0, +73434, -3432
```

4.2.6.2 Stowed Denotations

There are also denotations for arrays and structs. An array denotation has the form:

ARRAY (<element-list>) where all <element>s are denotations of the same type. An element which is itself an array need not repeat the word "ARRAY". A structure denotation has the form:

<type-name> (<denotation-list>)

where the type of the n'th denotation is the same as the type of the n'th field of structs of the named type.

Example:

```
ARRAY(3, 4, 5, 6)
ARRAY( (2,3) , (4,5) )
STRING (@name,size(name))
COMPL (3.045, -4.2)
```

4.2.6.3 Procedure Denotations

In ALGOL 68, there are no procedure denotations; rather, there are routine texts. These provide the actual values for procedures to possess. They are not called denotations because all other denotations are primal in scope, but a procedure has the scope of the range in which it is declared. Rather than change the semantics of "denotation", the authors of [34] chose another term. In ALGOL 68, the user cannot have procedure denotations. This is no mere quibble over niceties: creating procedure of primal scope is a major activity of systems programmers and PL370 has procedure denotations.

```
<procedure-denotation> ::= <linkage-indicant>  [ (
         <formal-parameter-list> )  ]   [ <result mode
         indicant> ]  : <serial-clause> CORP  [ <tag> ]
<formal-parameter-list> ::= <formal-parameters>  { ,
         <formal-parameters> }*
<formal-parameters> ::= <formal-parameter>  { , <tag> }*
      | <register-tag> <formal-parameter>
<formal-parameter> ::= LOCAL <formal-declarer> <tag>
      | [ VALUE ]  [ RESULT ]  <formal-declarer> <tag>
<routine-text> ::= <linkage-indicant>  [ (
         <formal-parameter-list> )  ]   [ <result mode
         indicant> ]  : <import-list> ; <serial-clause>
         CORP  [ <tag> ]
      | <procedure-denotation>
<import-list> ::= IMPORT <identifier>  { , <identifier> }*
```

Examples:

```
      S_TYPE (LONG REAL VALUE a,b) LONG REAL:
         a + b        CORP   add
      A_TYPE (INT VALUE num,width) STRING:
         IMPORT error char;
         ...
         CORP   whole
```

4.3 Declarations of Objects

4.3.1 Forward Declarations

Because everything must be declared before use, there must be a mechanism to allow the declaration of labels for forward branches and mutually recursive procedures.

```
<forward-object-declaration> ::= FWD <actual-mode-declarer>
         <identifier-list>
```

4.3.2 Procedures

The procedure declaration specifies the modes of any parameters passed in memory, and the body of the procedure. There are 3 kinds of procedures. An EXTERNAL procedure is declared in a source module, but its body is given in another source module. A GLOBAL procedure is declared in a source module with its defining body; it may be declared as EXTERNAL in other source modules. A LOCAL procedure is declared internally to a GLOBAL procedure. It may not be called except from inside that GLOBAL procedure. A LOCAL procedure may be a SEGMENT procedure, in which case it is independently addressed and does not inherit any of the data from the surrounding blocks. Other LOCAL procedures do not reload any base registers and do inherit the surrounding environment.

```
<procedure-declaration> ::= [ <scope> ] [
          <linkage-indicant> ]  <procmode-declarer>
          <proc-names> <initial-procedure-value>
      | [ <scope> ]  PROC <proc-names>
          <initial-procedure-value>
      | <linkage-indicant> <procmode-declarer> <proc-names>
<scope> ::= GLOBAL
      | EXTERNAL
      | SEGMENT
<proc-names> ::= <procedure-identifier> <external-names>
<initial-procedure-value> ::= <routine-text>
      | <procedure-identifier>
```

Examples:

```
      PROC(INT,INT) BOOL q = r

      PROC z = LOCAL INT: 3 CORP

      REF S_TYPE PROC ( A_TYPE PROC (REAL) REAL,
                      REAL ) REAL  func
```

4.3.3 Records

A record is simply a group of objects gathered together under one name. Like a COBOL record, it may also be used as if it were a simple ARRAY( )CHAR. Records may also be global or external.

```
<record-declaration> ::= [ GLOBAL | EXTERNAL ]  RECORD
          <identifier> <external-names> <record-body>
<record-body> ::= ( <extended-object-declaration-list> )
<external-names> ::= <emtpy>
      | / <string-denotation>  { , <string-denotation> }*
<extended-object-declaration-list> ::=
          <extended-object-declaration>  { ,
          <extended-object-declaration> }*
```

```
<extended-object-declaration> ::= <variable-declaration>
      | <constant-declaration>
      | CONS <mode-declarer> <denotation-list>
```

Examples:

```
      GLOBAL RECORD date/"DAT"
            (INT year, month, day, CHARS(8) weekday);
             ¢  a definition of named common  ¢

      EXTERNAL RECORD date/"DAT"
            (INT year, month, day, CHARS(8) weekday);
             ¢  a reference to named common  ¢

      RECORD header
         ( CHAR '1',
           CHARS(5) " ",
           CHARS(20) title,
           CHARS(40) " ",
           CHARS  "Page ",
           DISPLAYB(4) pn
         )
```

4.3.4 Operators

PL370 has many built-in operators, such as "+", "-", "<=". Operators behave like generic functions - there may be several definitions for the same operator and they will be distinguished by the types of the operands (actual parameters). An operator may be an indicant, a built-in operator, or a new operator. New operators are declared in the module header; e.g.:

```
      NEW ARITH OP: +, - .
      NEW COMP OP: ≥, ≤, ≠ .
      NEW BOOLEAN OP: ■ .
```

(See section 4.5.3 - Expressions.)

Sample declaration:

```
OP  CEIL  =  (REAL  x)INT: ( /* code to find the
value */ ) ;
```

4.3.5 Other Constants

Constants are declared using the word CONS. As in ALGOL the mode preceeds the identifier list. Each constant must be given a value when declared.

Examples:

```
CONS INT max int = 2147483647,
         lowest unit = 0, highest unit = 19

CONS REAL min real = REAL (
16r0000000000000001 )

CONS CHARS(8) LOGICAL_UNIT = ("SCARDS",
   "SPRINT", "SPUNCH", "GUSER", "SERCOM", "0",
   "1", "2", "3", "4", "5", "6", "7","8", "9",
   "10", "11", "12", "13", "14", "15", "16",
   "17","18", "19")
```

4.3.6 Other Variables

A variable may have an initial value specified. If the object code is requested to be re-entrant or reusable (see the option-list in the sample program), then the initial value is assigned to the variable at run-time.

```
<variable-declaration> ::= [ VAR ]  <mode-declarer>
          <var-decl> { , <var-decl> }*
<var-decl> ::= <identifier>
```

```
      | <identifier> := <value>
```

Examples:

```
      INT x := 3 /*  note  automatic  conversion  from
      whole to integer */
      INT x := 3, y, z := -2
      REAL z
      COMPL w
```

4.4 Scope Rules

The normal scope of an identifier is its enclosing block and all blocks contained by that block (standard ALGOL 60 scope rule). Because of the hardware on the IBM 370, only 4K bytes of memory can be addressed from any one register; therefore identifiers that might otherwise be usable may be unaddressable and thus invalid in certain contexts.

The programmer may restrict the use of variables by internal blocks. This is done in much the same way that access to files is restricted in MTS. Normally, any contained procedure may read and write a variable; this may be changed with the PERMIT compile-time operator. The PERMIT operator applies to the source lines statically following it.

Format:

```
PERMIT <what> { R | RW | NONE | W } <to what>
       [ , { R | RW | NONE | W } <to what> ]*
```

where <what> is a <variable> or a list of them, and <to what> is a <procedure identifier> or a list of them.

Examples:

```
PERMIT width R ALL
PERMIT size RW getsize, R OTHERS
```

In FORTRAN, a sub-program may refer to variables of only two ranges: global (COMMON) and local. In ALGOL, a procedure may refer to variables from ranges in between global and local as well. There are two things wrong with this. First, it is expensive to maintain the display necessary to allow addressing of the intermediate ranges. Second, misspellings in the inner range are not errors if the spelling used matches the spelling of an identifier declared in an outer range. Compare this with FORTRAN, in which there must be a declaration in the current sub-program before a variable can be non-local (i.e. COMMON). In PL370, the programmer may have the ALGOL features, which do have advantages, but he must work for them.

At the beginning of a procedure body, the programmer may use an IMPORT statement. This lists all those identifiers declared in the containing procedure (or IMPORTED into it) which this procedure needs. It may also specify "ALL" or "NONE". Those objects whose identifiers are on the import list will be made available to the procedure, and the identifiers may be used. If there is no import list, then all objects of primal scope will be imported. This includes named denotations and some procedures. Small constants will be copied into the local data area. For other objects, the address of the object will be placed in the local data area. Alternatively, the programmer may build a display by immediately basing "VARS OF <procedure-identifier>" for some procedures (see below). Those objects in the data area for the named procedures, and in the import list, will be made available and addressed from the named registers.

To address other non-local data, the programmer must tell the compiler where the data is. This is done with the BASE operator - it corresponds to the USING operator in IBMASM.

Format:

```
      BASE <data-area> ON <address>
```

where

```
<data-area> ::= <external-record-identifier>
      | VARS OF <procedure-identifier>
<address> ::= <register-name>
      | <register-name> + <whole-number>
      | <whole-number>
```

Examples:

```
      BASE savearea ON r13;

      IMPORT a,b,c,d,e, pq, maxint;
      BASE VARS OF pp ON r2;
```

The object then remains addressable until the containing clause ends, or until the programmer modifies any referenced registers.

4.5 Actions

4.5.1 Control Structures

A unit is either a statement (a void-unit) or an expression delivering a value of some MODE (a MODE-unit). A clause is a unit,optionally preceeded by statements, which are optionally preceeded by declarations (see below). The control structures in PL370 are almost the same as those in ALGOL68. With some syntax (below), the form of each control structure is now explained.

```
<clause> ::= <MODE-clause>  |  <void-clause>
<MODE-clause> ::= { <declaration> ; }* { <statement> ; }*
          <MODE-unit>
<void-clause> ::= { <declaration> ; }* { <statement> ; }*
          <void-unit>
<MODE-unit> ::= <MODE-expression>
<void-unit> ::= <statement>
```

The control structures are:

(1) <null statement> ::= SKIP | <empty>

(2) <block> ::=
BEGIN <clause> END |
(<clause>)

(3) <if clause> ::=
IF <boolean-clause> THEN <M-clause>
{ ELIF <boolean-clause> THEN <M-clause> }*
[ELSE <M-clause>]
FI

The <M-clause>'s must all be of the same mode, M.

(4) <indexed-case-clause> ::=
CASE <integer-clause>
IN <M-unit> { , <M-unit> }*
OUT <M-unit>
ESAC

All of the <M-unit>'s must be of the same mode, M, or they must all be void. The value of the <integer clause> is used to choose which of the <M-unit>'s to perform, counting from 1. If there is no such <M-unit>, then the <M-unit> after OUT is performed. " OUT SKIP " may be omitted.

(5) <labelled-case clause> ::=

```
      CASE  <M-clause>
      IN  <caselabels>  <N-unit>
        { , <caselabels>  <N-unit> }*
      OUT  <N-unit>
      ESAC
where  <caselabels>  ::=
      <M'-value> : { <M'-value> : }*
```

The <M-clause> delivers a value of a programmer-defined type,M. The <M'-value>'s are identifiers from the list defining the type M. All the <N-unit>'s are of the same mode, N, or they are all void. The <M-clause> is elaborated; The <N-unit> corresponding to the <M'-value> which matches the <M-clause> is performed.

```
(6) <conformity-case-clause>  ::=
      CASE  <tag>
      IN  (<M'-name> <other tag>):  <N-unit>
        { , (<M'-name> <other-tag>):  <N-unit> }*
      OUT  <N-unit>
      ESAC
```

Here <tag> is the identifier of some object of mode M, where M is UNION(...). Each <M'-name> is a <mode-name> for one of the constituent modes of the union. The <N-unit>'s are all of the same mode, N, or they are all void. The flag procedure for the union is elaborated to determine the current mode of the value of <tag>. The <N-unit> corresponding to the appropriate <M'-name> is elaborated. <other-tag> may be used to access the value of <tag> in the <N-unit>.

(7) While loops

```
   <basic-while-loop>  ::=
      [ { WHILE | UNTIL } <boolean-clause1> ]
      { DO | REPEAT }  <void-clause>
      [ { WHILE | UNTIL } <boolean-clause2> ]
      OD
```

The only required part of the loop is its <void-clause> body and its enclosing keywords. The loop repeats until one of the boolean tests is satisfied, i.e. a WHILE clause is FALSE or an UNTIL clause is TRUE. The boolean tests are made in the position that they appear - <boolean-clause1> is evaluated before the loop body and <boolean-clause2> after it. If <boolean-clause1> is omitted, then the loop will be elaborated at least once.

```
(7a) <incremental-while-loop>  ::=
      <iteration-control1> <basic-while-loop>
where <iteration-control1> ::=
      [ FOR  <tag>
        [ { FROM | := } <integer-unit> ]
        [ { UP | DOWN } ]
      [ BY  <integer-unit> ]     ]
     [ TO <integer-unit> ]
```

This introduces further constraints on the elaboration of the loop. If FOR <tag> appears, then that <tag> is known as a local CONS INT throughout the whole loop. The tag may be referenced in both the <boolean-clause>s and the <void-clause>. "FROM 1" and "BY 1" may be omitted. The <tag> (or secret loop counter if there is only a TO part) is incremented and tested at the end of the loop. The FROM part specifies the initial value for the <tag>. The BY and TO parts are evaluated only once. After each iteration the value of the BY part is added (subtracted, if DOWN is specified) to the control variable (<tag>). UP and DOWN assert at compile-time that the loop is ascending or descending, respectively; if neither is specified, then the direction of the loop is determined (at run-time, if necessary) by checking the sign of the BY part. E.g.:

```
      FOR r2 := 17 DOWN BY 1 TO 1 DO f(x(r2)) OD
```

```
(7b) <traversal-loop>  ::=
      <iteration-control2> <basic-while-loop>
where <iteration-control2>  ::=
      FOR  <MODE-name> <tag>
      [ FROM | := ] <MODE-unit>
      NEXT  <MODE-unit>
```

This is intended mainly for traversing lists. The unit after NEXT may reference the <tag>. The value of this reference is the value the control variable had during the iteration just finished; the value of the unit is the next value of the <tag>, i.e. the value of the <tag> during the next iteration of the loop. E.g.:

```
FOR REF TASK t := first OF tasklist OF tptr
 NEXT next task OF t
 UNTIL t = null OR ready to run flag OF t
 DO  SKIP
OD
```

(7c) <complete-loop> ::=

<iteration-control3> <basic-while-loop>

where <iteration-control3> ::=

FOR <tag> IN { <mode-name> | <multiple-value> }

This is intended mainly for accessing all values of a programmer-defined type. The <tag> will take each value of mode MODE.

```
FOR  s IN EGG_SIZE
 DO print( tag(s), minsize(s), maxsize(s), newline )
OD
```

(8) <exit> ::=

EXIT [[FROM] <label>] [WITH <MODE-unit>] [

]

IF <boolean unit>]

This construct jumps out of a labelled clause (see section 9 for an example). The <MODE-unit>, if any, becomes the value of the labelled clause; MODE must be the same as the MODE of the last unit of the labelled clause.

4.5.2 Units

4.5.2.1 Simple Expressions

All the basic actions from Section 3.15 are included; where several suggestions were made, the first one is the one used. On a less primitive level, the basic concepts of PL360 have been expanded in a similar manner.

The simple expression may be regarded as an abbreviation for a sequence of basic actions. From this point of view, the abbreviations are:

(1) long form: a := a OP^1 b OP^2 ...
    abbreviation: a OP^1:= b ; a OP^2:= ...
(2) long form: a := b := c ...
    abbreviation: b := c ... ; a := b

Also, some restrictions are removed :

- LONG REAL division will be simulated.

4.5.2.2 Object Designators

An object designator is a name or address . The object may be in memory or in a register.

```
<object-designator> ::= <identifier>
      | <denotation>
      | ( <object-designator> )
      | <field-name> OF <object-designator>
      | <object-designator> <sub> <field-name> <bus>
      | <object-designator> <sub> <subscripts> <bus>
      | <object-designator> ( <field-name> )
      | <object-designator> ( <subscripts> )
      | <object-designator> <index>
<subscripts> ::= <expression>  { , <expression> }*
<index> ::=  .  ( <expression : <expression> )
      | .  ( <expression | <expression> )
```

Examples:

```
      a
      re OF z
      (f2 OF w(35))(3)
      w(35)(f2)(3)  /* same as above */
```

4.5.2.3 Expressions

Most ALGOL 68 expressions are valid in PL370. The major exceptions are: no trimming, no balancing, no automatic widening, and no selections from [ ]REF STRUCT (...). There is no widening of BITS to [ ]BOOL because the two are semantically equivalent to start with. The rowing coercion will produce a one-element array with no descriptor.

```
<expression> ::= <object-designator>
```

```
        | <function-call>
        | <generator-indicant>  [ := <expression> ]
        | <formula>
        | BEGIN <serial-clause> END
        | ( <serial-clause> )
        | IF <boolean-clause> THEN <serial-clause> FI
        | IF <boolean-clause> THEN <serial-clause> <elses> FI
        | CASE <integer-clause> IN <expression-list> OUT
            <serial-clause> ESAC
        | CASE <serial-clause> IN <labelled-expression> { ,
            <labelled-expression> }* OUT <serial-clause> ESAC
        | CASE <tag> IN <conformity-expression> { ,
            <conformity-expression> }* OUT <serial-clause>
            ESAC
<function-call> ::= <procedure-identifier>
        | <procedure-identifier> ( <expression-list> )
<expression-list> ::= <expression> { , <expression> }*
<labelled-expression> ::= <case-labels> <expression>
<case-labels> ::= <identifier> : { <identifier> : }*
<conformity-expression> ::= <mode-indicant> { :
            <mode-indicant> }* : <expression>
        | ( <mode-indicant> <identifier> ) : <expression>
```

In a formula there are several different levels of precedence for operators. Monadic operators have the highest priority. Dyadic operators are divided into three classes: arithmetic, comparison, and boolean (in descending precedence). Within each of these three classes, expressions are evaluated from left to right.

4.6 Compilation Structures

A compilation consists of one or more source modules. A source module consists of an option-list, followed by the type, proctype, and mode declarations (if any), followed by declarations for one or more global objects (i.e.

procedures or records). An option-list tells the compiler what constraints to use, for example, what instruction set is available on the machine intended for running the object program. As in PL/I, each EXTERNAL object must be declared in each of the source modules that references it. All declarations of a global (shared between source modules) object must be the same.

Representation

All input is expected in EBCDIC. Normally there is no translation, but the programmer may request that the two cases of letters be equivalent (i.e., "a" = "A"). All keywords (control indicants, mode indicants, and operator indicants) are in upper case and are reserved. Since all mode declarations appear before any data declaration, there will be no difficulties in identifying which words are keywords or indicants even without case distinctions. An identifier is any combination of letters and digits that start with a letter (as in ALGOL), but may include both upper- and lower-case letters. Also, it may not be the same as any keyword, since they are reserved. To improve readability when only a single case is available, keywords and indicants may be prefixed with a ".", e.g. ".BEGIN". This does not make the un-prefixed word available as an identifier.

4.7 Extensibility Features

4.7.1 The Compile-time Language

In addition to the "ordinary" language, there is a compile-time language. This language is very similar to the ordinary language, but simpler. The keywords LINKAGE, ARRAYTYPE, REG, GET, RELEASE, OPEN, GENERATOR, and BASE are not available. ASSERT is available and behaves as in ALGOL W. Compile-time actions are distinguished by a trip character, "%". No code will be generated for any action which can be done at compile-time; in case of doubt, a run-time interpretation will be preferred. The compile-time language is interpreted during compilation of the ordinary language. The compile-time language has STRINGs as a basic mode. They are completely flexible and behave like strings in SNOBOL. There is only one length of integer (INT), one of whole (WHOLE), one of real (REAL), and one of pointer (REF ...); the other basic modes are BOOL and CHAR. Since types have been made superflous (there is only one mode per type), the extra concept has been dropped. Programmer-defined modes are still available. The only kind of arrays are ROW with a full descriptor, as in ALGOL 68. There are no input facilities, although 'parstring' (the parameter from the system) may be inspected. There are all the ALGOL 68 unformatted output facilities except FILE declarations; there are two-predefined files. The programmer may write on the listing via "print", and may generate source text via "gen". The compile-time language allows arbitrarily complex expressions, as in ALGOL 68. The compile-time language does not provide access to registers or allow privileged operations.

4.7.2 Macros

There are several kinds of macros. There are text macros (called MACRO) and then there are PROCs and OPs which are compiled in-line (called IPROCs and IOPs). The header of a macro consists of keywords and symbols to be entered verbatim in a call, and parameter-packs.

```
<macro-definition> ::= MACRO <header> ; <text> MEND
<header> ::= [ <exposed-macro-formal-parameter-pack> ]
          <closed-header> [
          <exposed-macro-formal-parameter-pack> ]
<closed-header> ::= <macro-delim>
      | <closed-header> <mfpl-open>
          <macro-formal-parameter-list> <mfpl-close>
          <macro-delim>
<exposed-macro-formal-parameter-pack> ::= <mfpl-open>
          <macro-formal-parameter-list> <mfpl-close>
<macro-formal-parameter-list> ::= <macro-formal-parameter> [
          , <macro-formal-parameter> ]*
<macro-delim> ::= <actual-text-to-match-for>
<macro-formal-parameter> ::= <identifier>
<mfpl-open> ::=  <
<mfpl-close> ::=  >
```

A special case of this is the procedure notation wherein the first <macro-delim> is the name of the macro followed by a "(" and the second and last <macro-delim> is a ")". To call a macro, the <macro-delim>s are given exactly as in the definition (except for numbers of spaces, as in identifiers), the angle brackets are replaced by spaces, and the <macro-formal-parameter>s are replaced by source-text expressions. In the case of an <exposed-macro-formal-parameter-pack>, the actual parameters are restricted to single-token expressions, i.e. a identifier, or a denotation of one of the basic types, a string, an indicant, or an operator.

Sample definition:

```
¢ This macro subscripts a matrix
¢
MACRO <vec> [ <i,j> ] ;

  (IF i < 1 LWB vec  OR  j < 2 LWB vec
    THEN  GO TO  lwberror  FI;
   IF i > 1 UPB vec  OR  j > 2 UPB vec
    THEN  GO TO  upberror  FI;
   mem ( @vec + ((i - (1 LWB vec))*(2 DIM vec)
         +  (j - (2 LWB vec))) * element
size(vec) )
  )
   MEND
```

Sample call:

```
u OF x [ 7*i-lkf%3•km, h(u OF j,-345) ]
```

Sample expansion:

```
u OF (IF  7*i-lkf%3•km < 1 LWB x
          OR h(u OF j,-345) < 2 LWB x
       THEN  GO TO  lwberror  FI;
      IF  7*i-lkf%3•km > 1 UPB x
          OR h(u OF j,-345) > 2 UPB x
       THEN  GO TO  upberror  FI;
      mem ( @x + ((7*i-lkf%3•km -  1  LWB  x)*(2
          DIM x)
            +  (h(u  OF  j,-345)  -  2 LWB x)) *
          element size(x)
     )
```

4.8 Run-time Organization

The programmer has his choice of several storage organizations. He makes this choice by using some of the options at the head of the source module. The compiler will always produce one CSECT per global procedure. It should produce one object module per source module under OS and one object module per global object under MTS; this is because of the different capabilities of the linkage editors in these systems.

Constant data will be located with the code in the order of their declarations. They will normally be moved after the nearest unconditional branch, or nearer if that can shorten the object code. The programmer may request that the constants be gathered in larger groups: all the constants for any procedure, a segment procedure, or a global procedure to be placed before or after the code.

Variables will normally be located in areas dynamically acquired at the beginning of each global procedure (re-entrant). The variables may be acquired instead at the beginning of every procedure by using the same linkage type for internal procedures as for the global procedure; this in itself does not give the programmer the ability to use recursive procedures - it is only a necessary condition. The variables may be located in a static location, i.e. a separate CSECT, by specifying "re-usable". The variables can also be allocated in the same way as the constants by specifying "non-reusable".

CHAPTER 5

IMPLEMENTATION

5.1 Desired Environment for Program Design

5.1.1 Compiler Services

The burden of a PL370 compiler is more than just producing the best possible object code from the source code. It must be capable of:

- reading the input from any type of file
- ignoring case distinctions in the source while showing them in the listing
- producing a good listing, including
    - the complete input line
    - a full header, with the defaults filled in
    - paragraphing the input neatly
    - showing the object separately or beside the source (user's choice)
    - supporting titles and spacing controls
    - optional expansion of macros
    - an optional cross-reference
    - an optional alphabetical list of tags and their

complete modes (like the PL/I ATTR table)

- producing warnings for non-use of computed values
- catching operations that change the mode of an object without permission
- catching all violations of choices specified in the module header
- generating code for debugging purposes (e.g. actually check assertions)
- allowing macros, pieces of source text, and environment descriptions to be kept in libraries and referenced from the program, rather than requiring everything to be kept in the module source file.
- producing SYM records for the operating system debugging system
- supporting separate compilation as described in Chapter 3.

There must be a standard library of procedure declarations to interface with the operating system. Under OS, these would issue the standard supervisor calls properly; under MTS, they would just make the present routines known to the compiler.

The compiler must also include much more environment information in the object output than the usual case, to allow the linkage editor to ensure that the linkage is correct.

A useful feature of the IBM Assembler is that one can always get an object module. Frequently, source errors can be corrected and the object module used with few or even no changes (saving re-assembly). Therefore, the compiler should always strive for some plausible interpretation and produce code for that.

5.1.2 Register Management

The compiler must carefully trace register usage. It must produce errors warning of those cases when:

- a value in a register is lost without use or explicitly being discarded (when a value includes several register, then only a part of it need be used)
- a register in local use at the time of a call is needed for parameter passing
- a register in local use at the time of a call is one that is used for result passing or is not restored
- a register is used without being acquired via GET

Initially all registers are under the control of the programmer. To allow maximum flexibility, he must immediately RELEASE them all. When the compiler controls no registers, then expressions are severely restricted and many features are unavailable, e.g.:

`<REAL mem> +:= <REAL mem>`

5.1.3 Run-time Support

In order to ease programming, there should be available some libraries of routines to do transput. This should include routines to convert string input to internal form and internal data back to string. It should also include routines to do simple stream transput as in ALGOL 68. Other routines in the library will be LONG REAL division, LONG forms of translate (TR) and scan (TRT), and LONG forms of move (MVCL) and compare (CLCL) for programs compiled to run on /360's.

There will also be some routines required (despite the goal of no required run-time routines) to report errors. For example, if subscripts are being checked, then "subscript error" will be called if an error occurs. A routine "null proc" will be called when a PROC variable contains an invalid address (e.g. NIL) and that "proc" is called.

5.2 Outline Of A Compiler

A two-part compiler is envisioned. A pre-processor will perform all the compile-time actions, and a processor will generate code from the resulting source text. Either one may be used without the other. Because everything is declared before being used, a one-pass processor (with fixup) is possible. The pre-processor can also be done in one pass, as it consists of interpreting the input as a compile-time language program. If no built-in pre-processor primitive gives information about text not yet read when that primitive is first encountered, then the pre-processor can produce source text in one pass without even needing any fixup. In this case the two parts can be run as co-routines and appear as one pass to the user. This restriction however, does prohibit some forms which were proposed earlier. A problem is the proposed primitive "calls", which is TRUE if the procedure calls any procedure. If this primitive is kept, then the pre-processor must use two or more passes. If not, then calling will be slightly less efficient, or there must be two different S-type linkage definitions, one which allocates a save area and may call, and one which doesn't and can't. This works, but not cleanly, as the call part of a linkage definition describes how that class of procedures is called, not how it calls; so the unwanted call will be detected only because, at the call, register 13 is not pointing at a writeable save area.

The environment descriptions that the compiler produces can be modelled after those of ALGOL68C [8].

It is very difficult to estimate implementation costs. Using a good compiler-writing system, one or two men might finish a complete PL370 system in under a year. If one wanted a super compiler which produced optimal code (i.e. significantly better than the average assembler programmer), then it might take an extra year or more.

CHAPTER 6

CONCLUSIONS

6.1 What Has Been Accomplished

This thesis has proposed a solution to a major problem in systems programming. Twelve years after the introduction of the IBM 360, applications programmers have been given several suitable languages, but systems programmers are still waiting for some language which really is an improvement over Assembler.

Assembly language and eleven other languages were discussed in detail, examining the reasons why each of them is not good enough for producing the reliable systems that are needed. Some languages are error-prone, others are not properly implemented, and others are just too expensive. While they all can be used for writing self-contained systems, such as APL and SPSS, none of the high-level languages can be used to write service routines to be called from any other high-level language.

Service routines are needed for run-time systems for other languages and operating systems. A new language, PL370 has been proposed. It is capable of being a general-purpose assembler replacement and also in being a good language for the production of reliable service routines. Many languages have been proposed which were designed for the writing of compilers. Also, it has become fashionable to demonstrate a language by showing a compiler for it, written in itself. On the other hand, PL370 is the first language designed for the writing of run-time systems and is not recommended for compiler writing.

PL370 is based on ALGOL 68, but other language have influenced its design. Two important influences were PL360 and COBOL. The former was an example of where the language design was heading. The latter was used because I believe that the liberal use of English keywords can result in more readable, hence more reliable, programs.

One can use PL370 from at least three different approaches. First, one can try to ignore the differences between it and ALGOL 68 as much as possible and have a general-purpose language with almost no run-time system. Secondly, one can take care of as much register allocation as one wants and use all the instructions available on the machine to have a medium-level machine-dependent language. Thirdly, one can specify each instruction explicitly as in a low-level language.

6.2 Future Work

It is my opinion that PL370 is worthy of future work. The next step should be the building of a compiler for the language as defined herein. This should be a table-driven compiler so that it can be written and changed easily and quickly. Then a group of systems programmers should be given the compiler to use for a test period of from three to twelve months. After this time, the language design should be revised in the light of the criticisms made the users. Then a high-quality production compiler should be built. If there are enough users, an optimizing compiler should also be built.

Bibliography

1. ANSI : "American National Standard X3.9-1966-FORTRAN", American National Standards Institute, New York, New York, U.S.A. (1966)

2. CODASYL : COBOL Journal of Development, printed by Government of Canada, Dept. of Supply and Services, Ottawa, Ontario (1973)

3. IBM: "OS Assembler Language", Form GC28-6514, International Business Machines (1970)

4. IBM : "PL/I(F) Language Reference Manual", Form C28-8201, International Business Machines

5. IBM : "System 360 Principles of Operations", Form GA22-6821, International Business Machines (1968)

6. IBM : "System 370 Principles of Operations", Form GA22-7000, International Business Machines 1974)

7. H. Boom : private communication (1974)

8. S. R. Bourne, A. D. Birrell, I. Walker : "ALGOL68C Reference Manual", Preliminary Edition, University of Cambridge Computing Service, Cambridge, U. K. (1975)

9. B. L. Clark, F. J. B. Ham : "The Project SUE System Language Reference Manual", Technical Report CSRG-42, Computer Systems Research Group, University of Toronto, Toronto, Ontario (1974)

10. R. Conradi, P. Holager : "A study of MARY's data types in a systems programming application", in [32] below, pp. 295-309.

11. J. B. Dennis : "An Example of Programming with Abstract Data Types", Sigplan Notices 10,7 (1975)

12. E. W. Dijkstra : A Discipline of Programming, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, U.S.A. (1976)

13. E. W. Dijkstra : "Go to Considered Harmful", CACM 11,3 (1968)

14. J. D. Gannon : "Language Design to Enhance Programming Reliability", Technical Report CSRG-47, Computer Systems Research Group, University of Toronto, Toronto, Ontario (1975)

15. C. Gray : *ALAI: A Language for Artificial Intelligence*, Master's Thesis, Department of Computing Science, The University of Alberta, Edmonton, Alberta (1976)

16. A. N. Habermann : "Critical Comments on the Programming Language Pascal", Acta Informatica, Vol. 3, Fasc. 1, pp. 47-57 (1973)

17. C. A. R. Hoare : "Hints on Programming Language Design", STAN-CS-73-403, Computer Science Department, Stanford University, Stanford, California, U.S.A. (1973)

18. B. Kernighan : "Programming in C - A Tutorial", Bell Labs (197?)

19. B. Kernighan : "UNIX for Beginners", Bell Labs (197?)

20. D. E. Knuth : "Structured programming with go to Statements", Computing Surveys, 6,4 (1974)

21. L. Leader, R. Salisbury (eds.) : *Michigan Terminal System, Volume 3 - Subroutine and Macro Descriptions*, Third Edition, revised, The Univeristy of Michigan Computing Center, Ann Arbor, Michigan, U.S.A. (1973)

22. O. Lecarme, P. Desjardins : "Reply to a Paper by A. N. Habermann on the Programming Language Pascal", Sigplan Notices 9,10 (1974)

23. M. Malcolm : "PL360 (Revised) A Programming Language for the IBM 360", STAN-CS-71-215, Computer Science Department, Stanford University, Stanford, California, U.S.A. (1972)

24. P. Naur (ed.) : "Revised Report on the Algorithmic Language ALGOL 60", CACM 6,1 (1963)

25. J. Pearkins, *FMT Reference Manual*, Computing Services, The University of Alberta, Edmonton, Alberta (1976)

26. M. Rain : "MARY Programmers' Reference Manual", RUNIT, Trondheim, Norway (1973)

27. M. Rain : "Some Formal Language Aspects of MARY" or "Algol X Revisited", SINTEF, Trondheim, Norway (1972)

28. M. Rain : "A possible resolution of the subroutine calling reference problem in machine oriented languages", in [32] below, pp. 193-206

29. M. Richards : "The BCPL Programming Manual", The Computer Laboratory, University of Cambridge, Cambridge, U. K. (1973)

30. D. Ritchie : "C Reference Manual", Bell Labs (197?)

31. R. L. Sites : "ALGOL W Reference Manual", STAN-CS-71-230, Computer Science Department, Stanford University, Stanford, California, U.S.A. (1971)

32. W. L. van der Poel, L. A. Maarssen (eds.) : Machine Oriented Higher Level Languages, North-Holland Publishing Company, Amsterdam (1974)

33. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck : "Draft Report on the Algorithmic Language ALGOL 67", MR88, Mathematische Centrum, Amsterdam, Netherlands (1967)

34. A. van Wijngaarden, et al. : "Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica, Vol. 5, Fasc. 1-3 (1975)

35. D. Webster (ed.) : MTS Command Language, Computing Services, The University of Alberta, Edmonton, Alberta (1976)

36. N. Wirth : "A Programming Language for the 360 computers", JACM 5,1 (1968)

37. N. Wirth : "The Programming Language Pascal", Acta Informatica, 1a (1971)

38. N. Wirth : "The Programming Language Pascal (Revised Report)", Berichte der Fachgruppe* Computer Wissenschaften, Eidgenoessische Technische Hochschule, Zurich, Switzerland (1972)

39. W. S. Wulf, D. B. Russell, A. N. Habermann : "Bliss: A Language for Systems Programming", CACM 14,12 (1971)

40. W. Wulf, C. Geschke, D. Wile, J. Apperson : "Reflections on a Systems Programming Language", Proceedings of a Sigplan Symposium on Languages for Systems Implementation, in Sigplan Notices, 6,9 (1971)

41. C. T. Zahn : "Structured Control in programming Languages", reprinted from AFIPS Conference Proceedings, Vol. 44, Proceedings of the 1975 NCC, in Sigplan Notices, 10,7 (July,1975)

APPENDIX I

Moving and Adding in IBMASM

The IBM 370 [6] has many instructions which just move data to, from, or within the processor and memory. Some of these instructions set the condition code, but this side effect is ignored in this appendix. These are:

BR	exact copy
BALR	exact copy
ED	merges for copy
IC	exact copy
ICM	exact copy
IPK	exact copy
ISK	exact copy
LR	exact copy
L	exact copy
LCTL	exact copy
LH	modified copy - sign is extended
LM	exact copy
LPSW	exact copy
MVI	exact copy
MVC	exact copy, except when overlap
MVCL	copy may be padded
MVN	exact copy
MVO	exact copy
MVZ	exact copy
PACK	modified copy - deletes zones
RDD	exact copy
SCK	exact copy
SCKC	exact copy
SPT	exact copy
SPX	exact copy
SPM	exact copy
SPKA	exact copy
SSK	exact copy
SSM	exact copy
ST	exact copy
STIDC	exact copy
STC	exact copy
STCM	exact copy
STCK	exact copy
STCKC	exact copy
STCTL	exact copy
STAP	exact copy
STIDP	exact copy
STPT	exact copy
STH	exact copy
STM	exact copy
STPX	exact copy
UNPK	modified copy - inserts zones
WRD	exact copy
ZAP	modified copy - may add leading zeroes

LDR	exact copy
LD	exact copy
LER	exact copy
LE	exact copy
STD	exact copy
STE	exact copy

---

51 instructions   of which 44 are exact copying instructions

APPENDIX II

Sample Linkage Declarations

II.1 IBM S-type Linkage

```
LINKAGE  S_TYPE =

INPUT  (PTR CONS PALIST  r1,
        PTR SAVEAREA  r13,
        RETURN  r14,
        ENTRY  r15)
,
OUTPUT ( INT return code == r15 )
,
NORESTORE  (f01, f23, f45, f67)
,
SAVEAREA = S_TYPE_SAVE
,
CALL =
   %PROC(plist):
   << BEGIN >>
   INT  np = UPB parameter;
   IF  np > 0
    THEN
      PROC  parm = (INT i):
         IF  known(i)
          THEN  <<  @parameter(i)  >>
         FI
         CORP;
      BOOL any := FALSE;
      FOR i  TO np  WHILE ¬any
       DO  any := known(i)  OD;
      <<  /* a particular ARRAY (np) ADDRESS  */
          PALIST  parm list  >>
      IF  omitted(plist)
       THEN
       ELSE
         <<  ==  plist  >>
      FI;
      IF  any
       THEN
         BOOL all := TRUE;
         FOR i  TO np  WHILE all
          DO  all := known(i)  OD;
         IF  omitted(plist) AND all
```

```
          THEN  << == >>
          ELSE  << := >>
         FI;
         <<  (  parm(1)  >>
         FOR i  FROM 2  TO np
          DO  <<  ,  parm(i)  >>  OD;
         IF  known(np)
          THEN  <<  OR  16r80000000  >>  FI;
         << )  >>
      FI ;
      << ; >>
      FOR i  TO np
      DO
         IF  ¬known(i)
            <<
            select field(parm list,i) := @parameter(i);
            >>
         FI;
      OD;
      IF  ¬known(np)  THEN
         <<  select field(parm list,np).(0|1) OR:= 16r80; >>
      FI;
      <<  r1 := @parm list;  >>
   FI;
   <<  CALL  procname
       END  >>
   CORP
,
ENTER =
   %PROC(base):
   /*  implicit GET for each INPUT register  */
   <<  BEGIN
       SAVE  regsave OF r13 := (r14,...,r12);
       RELEASE  r14;  >>
   IF  base ¬= r15
    THEN
      TEXT  br;
      IF  omitted(base)
       THEN  br := genlab;  <<  REG  >>
       ELSE  br := base;    <<  GET  >>
      FI;
      <<  REF INSTRUCTION  br = r15;
          BASE CODE ON br;
          RELEASE  r15;  >>
   FI;
   IF  UPB parameter > 0
    THEN  <<  REG REF PALIST  plist = r1;   RELEASE  r1;  >>
   FI;
   IF  reentrant
    THEN
      IF  calls
       THEN
         <<  getspace(r0=3,r1=size(LOCVARS)+72);
             next OF r13 := r1;
             SAVE  previous OF r1 := r13;
             r13 := r1;
             BASE  VARS  ON  r13+72;
         >>
       ELSE
         <<  getspace(r0=3,r1=size(LOCVARS));
             REG REF LOCVARS db := r1;
             PERMIT r13 R
             BASE  VARS  ON  db;
         >>
      FI
    ELSE
      IF  calls
       THEN
         <<  BASE VARS AFTER CONS+size(SAVEAREA);
             (REG ADDRESS  t = @VARS-size(SAVEAREA);
              ASSERT  t  IS  REF SAVEAREA;
              next OF r13 := t;
              SAVE previous OF t := r13;
              r13 := t)  >>
       ELSE
         <<  PERMIT r13 R
             BASE VARS AFTER CONS;  >>
      FI
   FI
   CORP
;
EXIT =
   %PROC(INT rc):
   <<  ;  >>
   IF  calls
    THEN
      IF  reentrant  THEN   <<  r1 := r13;  >>  FI;
      <<  RESTORE  r13 := previous OF r13;  >>
      IF  reentrant  THEN  <<  freespace(r0=0,r1);  >>  FI
   FI;
   <<  RESTORE  r14 := (regsave OF r13)(1);  >>
   CASE  result
   IN  VOID: <<  RESTORE  r0 := (regsave OF r13)(3)  >>,
       GENRL: << r0 := result  >>,
       LONG INT: << d0 := result  >>,
       LONG REAL: << (RESTORE  r0 := (regsave OF r13)(3);
                      lfr0 := result)  >>,
       REAL: << (RESTORE  r0 := (regsave OF r13)(3);
                 fr0 := result)  >>,
       SHORT REAL: << (RESTORE  r0 := (regsave OF r13)(3);
                       sfr0 := result)  >>
   OUT  error("can't return " + result)
   ESAC;
   << ;  RESTORE  (r1,...,r12) := (regsave OF r13)(4:15)
      END  >>
   CORP
;
```

II.2 ALGOL68C Linkage

```
MODE  A_TYPE_SAVE =
   STRUCT (GENRL  saver0,
           RETURN saver1,
           ADDRESS saver2,
           REF INSTRUCTION saver3, saver4, saver5 );

LINKAGE  A_TYPE =

INPUT (RETURN  r1,
       REF GDSECT  r13,        ¢ MODE GDSECT not defined here
       REF SAVEAREA  r14,
       ENTRY  r15)
,
NORESTORE  (f01, f23, f45, f67, r6, r7, r8, r9, r10, r11)
,
SAVEAREA = A_TYPE_SAVE
,
CALL =
   %PROC :
   INT  np = UPB parameter;
   <<  @DOUBLE  >>
   IF  np > 0
    THEN
      << CONS PLIST  parm list = ( parameter(1)  >>
      FOR i  TO np   DO  <<  ,  parameter(i)  >>  OD;
      << ); >>
    ELSE
      << ¢  We need the address of the next available byte
         ¢ on the stack (for the callee to use (see below).
         ¢  This 'parm list', of course, is NOT really a
         ¢ parameter list.
         BYTE parm list;  >>
   FI;
   <<  r14 := @parm list;
       CALL  proc name
       END  >>
   CORP
,
ENTER =
   %PROC :
   IF  ¬reentrant  THEN  error  FI;
   <<  BEGIN
       /*   register  synonyms    */
       ADDRESS  plink == p1, pp == r2,
                base == r3, newp == r14;
       OPEN  r13;
       SAVE  reg save OF r14 := (r0,...,r5);
       GET REF INSTRUCTION r3 = r15;
       BASE CODE ON r3;     RELEASE  r15;
       GET ADDRESS  r2 = r14;
       RELEASE  r14;
       BASE DATA ON r2;
   >>
   IF  UPB parameter > 1
    THEN
      <<  /*  first 'data' is the parameter list    */
          /*  Reserve it now  */
          PLIST  parm list;
          OPEN  parm list;  >>
   FI
   CORP
,
EXIT =
   %PROC :
   CASE  result
   IN
      VOID:  <<  SKIP   >>,
      INT:  <<  GET INT r6 := value   >>,
      CHAR:  <<   GET CHAR r6 := value   >>,
      BITS:  <<   GET BITS r6 := value   >>,
      BOOLEAN(1,0):  <<   GET BOOLEAN(1,0) r6 := value   >>,
      ADDRESS:  <<   GET ADDRESS r6 := value   >>,
      REAL:  <<   GET REAL fr2 := value   >>,
      BYTES:  <<    GET BYTES fr2 := value  >>
   OUT  error
   ESAC;
   <<  ;  RESTORE  (r0,...,r5) := save regs OF r2
       END  >>
   CORP
;
```